

# Trabajo Práctico N°1

**Materia:** Arquitectura de Software [TB054]

## Profesores:

- Christian Calonico
- Mariano D'Ascanio
- Guillermo Rugilo

## Integrantes:

- 106895, Federico Solari Vazquez
- 108313, Rafael Ortegano
- 108091, Martin Morilla
- 109585, Gian Franco Keberlein

**Fecha de entrega:** 3 de Octubre de 2024



# Índice

<b>1</b>	<b>Introducción</b>	<b>3</b>
<b>2</b>	<b>Objetivos</b>	<b>3</b>
<b>3</b>	<b>Tácticas</b>	<b>4</b>
3.1	Guía de iconos . . . . .	5
3.2	Diagrama de componentes . . . . .	6
3.2.1	Táctica: Caso Base . . . . .	6
3.2.2	Táctica: Caché . . . . .	7
3.2.3	Táctica: Rate Limiting . . . . .	7
3.2.4	Táctica: Replicacion . . . . .	8
3.3	Hipótesis acerca de las tácticas . . . . .	8
3.4	Descripción de las arquitecturas . . . . .	8
3.4.1	<u>Caso base</u> . . . . .	8
3.4.2	<u>Caché</u> . . . . .	9
3.4.3	<u>Rate Limiting</u> . . . . .	9
3.4.4	<u>Replicación</u> . . . . .	9
<b>4</b>	<b>Herramientas de mediciones</b>	<b>9</b>
4.1	Artillery . . . . .	10
4.1.1	¿Qué es y cuál es su objetivo? . . . . .	10
4.1.2	¿Por qué Artillery? . . . . .	10
4.1.3	¿Dónde utilizamos Artillery? . . . . .	10
4.1.4	Pruebas de Carga . . . . .	11
4.1.5	Pruebas de Estrés . . . . .	11
4.1.6	Diferencias Clave . . . . .	11
4.1.7	Nuestros escenarios . . . . .	11
4.2	¿Cómo visualizamos nuestro programa a lo largo del tiempo? . . . . .	13
<b>5</b>	<b>Ejecución del programa</b>	<b>16</b>
<b>6</b>	<b>Nuestras mediciones</b>	<b>19</b>
6.1	Caso base . . . . .	19
6.1.1	Escenario: Carga , Endpoint: /ping . . . . .	19
6.1.2	Escenario: Carga , Endpoint: /facts . . . . .	20
6.1.3	Escenario: Carga , Endpoint: /dictionary . . . . .	22
6.1.4	Escenario: Carga , Endpoint: /spaceflight_news . . . . .	23
6.1.5	Escenario: Estres , Endpoint: /ping . . . . .	25
6.1.6	Escenario: Estres , Endpoint: /facts . . . . .	26
6.1.7	Escenario: Estres , Endpoint: /dictionary . . . . .	28
6.1.8	Escenario: Estres , Endpoint: /spaceflight_news . . . . .	29
6.2	Caché . . . . .	31
6.2.1	Escenario: Carga , Endpoint: /ping . . . . .	31
6.2.2	Escenario: Carga , Endpoint: /facts . . . . .	31
6.2.3	Escenario: Carga , Endpoint: /dictionary . . . . .	31
6.2.4	Escenario: Carga , Endpoint: /spaceflight_news . . . . .	33
6.2.5	Escenario: Estres , Endpoint: /ping . . . . .	34

6.2.6	Escenario: Estres , Endpoint: /facts . . . . .	34
6.2.7	Escenario: Estres , Endpoint: /dictionary . . . . .	34
6.2.8	Escenario: Estres , Endpoint: /spaceflight_news . . . . .	36
6.3	Rate Limiting . . . . .	38
6.3.1	Escenario: Carga , Endpoint: /ping . . . . .	38
6.3.2	Escenario: Carga , Endpoint: /facts . . . . .	39
6.3.3	Escenario: Carga , Endpoint: /dictionary . . . . .	41
6.3.4	Escenario: Carga , Endpoint: /spaceflight_news . . . . .	42
6.3.5	Escenario: Estres , Endpoint: /ping . . . . .	44
6.3.6	Escenario: Estres , Endpoint: /facts . . . . .	45
6.3.7	Escenario: Estres , Endpoint: /dictionary . . . . .	47
6.3.8	Escenario: Estres , Endpoint: /spaceflight_news . . . . .	48
6.4	Replicación . . . . .	49
6.4.1	Escenario: Carga , Endpoint: /ping . . . . .	49
6.4.2	Escenario: Carga , Endpoint: /facts . . . . .	51
6.4.3	Escenario: Carga , Endpoint: /dictionary . . . . .	53
6.4.4	Escenario: Carga , Endpoint: /spaceflight_news . . . . .	55
6.4.5	Escenario: Estres , Endpoint: /ping . . . . .	57
6.4.6	Escenario: Estres , Endpoint: /facts . . . . .	59
6.4.7	Escenario: Estres , Endpoint: /dictionary . . . . .	61
6.4.8	Escenario: Estres , Endpoint: /spaceflight_news . . . . .	63
<b>7</b>	<b>Conclusiones</b>	<b>65</b>
7.1	/ping . . . . .	65
7.2	/facts . . . . .	66
7.3	/dictionary . . . . .	66
7.4	/spaceflight_news . . . . .	66

## 1 Introducción

En el presente informe se describen y comparan distintas tácticas arquitectónicas aplicadas a una aplicación basada en Node.js. Dicha aplicación funciona como un servicio HTTP que consume diversas APIs externas para entregar información a los usuarios. Las tácticas implementadas incluyen:

- Caso base
- Caché
- Replicación
- Rate-Limiting

Cada una de estas tácticas se implementó de manera individual, manteniendo el núcleo funcional de la aplicación, para luego evaluarlas y compararlas en función de atributos de calidad clave, como el rendimiento (Performance) y la disponibilidad (Availability).

Adicionalmente, se diseñaron distintos escenarios de carga personalizados para someter a prueba las diferentes arquitecturas, permitiendo así un análisis detallado de sus fortalezas y debilidades bajo diferentes condiciones operativas.

## 2 Objetivos

Nuestro objetivo principal es comparar diferentes tecnologías y analizar cómo ciertos factores impactan en los atributos de calidad del sistema, identificando posibles mejoras. Además, buscamos familiarizarnos con el uso de diversas herramientas modernas y ampliamente utilizadas en la industria, tales como:

- Node.js (+ Express)
- Docker y Docker Compose
- Nginx y Redis
- Generadores de carga (como Artillery)
- Herramientas para medición y visualización en tiempo real, como cAdvisor, StatsD, Grafana y Graphite.

### 3 Tácticas

Como se menciono anteriormente, llevaremos a cabo 4 tacticas. Para implementarlas, empezamos creando un repositorio en GitHub, en el cual se pueden visualizar las 4 ramas donde se implemento cada arquitectura.



Cada rama en nuestro repositorio tiene su tactica correspondiente (sin contar la rama main), y son independientes entre si. Es decir, no se mezclan tacticas.

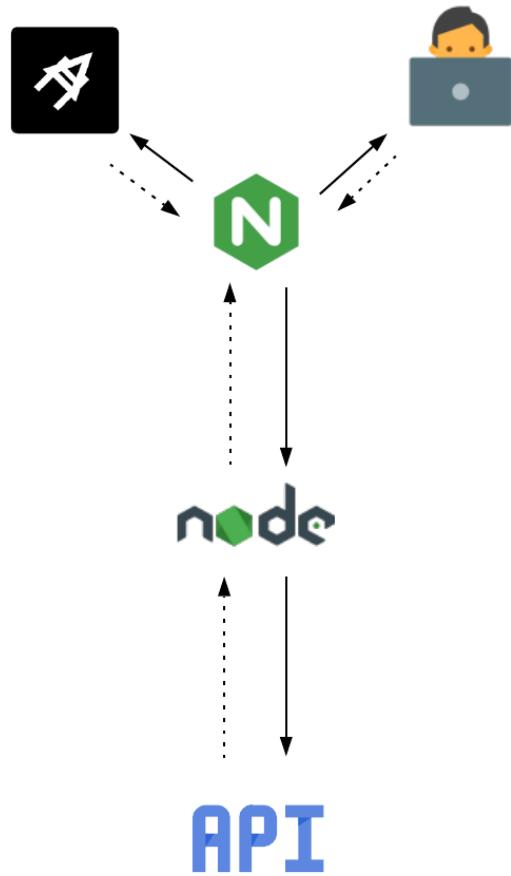
De forma que el lector pueda visualizar que es a lo que nos referimos con *tacticas/arquitecturas*, dejaremos unas imagenes que lo ejemplifican.

### 3.1 Guía de iconos

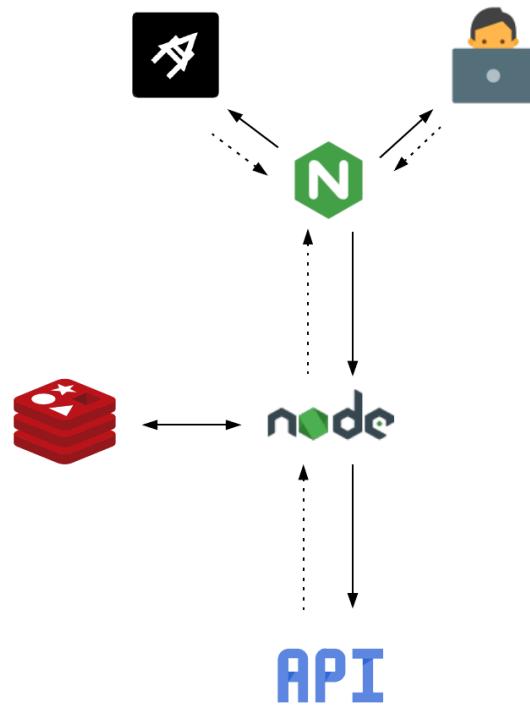
Icon	Reference
----->	HTTP Response
----->	HTTP Request
<----->	Redis serialization Protocol
	Node JS Server
	Redis cache
	Nginx Reverse Proxy
	Remote API
	Artillery Testing Platform
	LocalHost
	Request limiter per ip

### 3.2 Diagrama de componentes

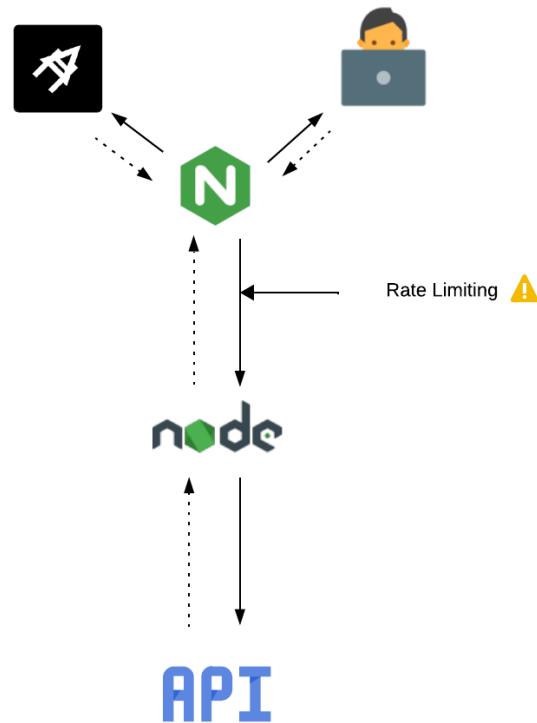
#### 3.2.1 Táctica: Caso Base



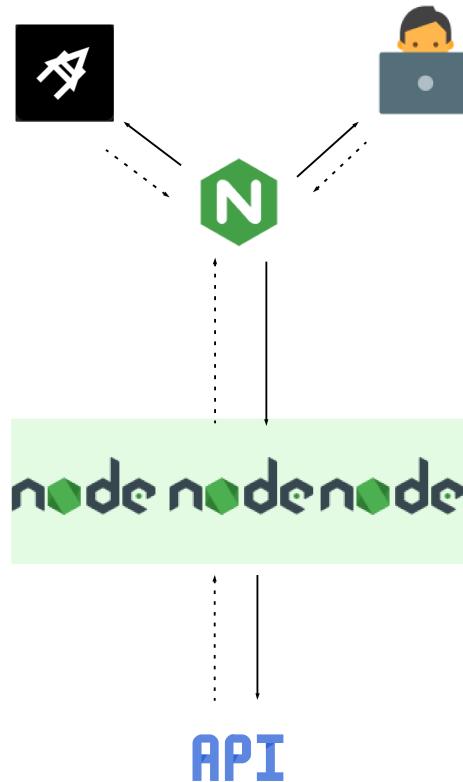
### 3.2.2 Táctica: Caché



### 3.2.3 Táctica: Rate Limiting



### 3.2.4 Táctica: Replicacion



## 3.3 Hipótesis acerca de las tácticas

Como se puede observar, cada arquitectura tiene algo que la hace diferente a la otra. En el entorno del desarrollo web, una de las principales problemáticas al iniciar un proyecto es determinar qué decisiones arquitectónicas serán más convenientes para no afectar negativamente a los atributos de calidad. Es verdad que muchas veces existe un "trade off" entre ellos, pero nosotros como desarrolladores buscamos que haya un equilibrio. Los desarrolladores suelen buscar lo que consideran 'mejor', sin tener en cuenta que no siempre existe una única solución ideal para todos los casos. La elección de ciertos componentes, como el uso de caché o la adopción de patrones arquitectónicos específicos, puede impactar significativamente los atributos de calidad, como el tiempo de respuesta, la capacidad de escalado o la mantenibilidad del sistema. Por lo que, algunas preguntas que el lector podría hacerse pueden ser: ¿Será que una táctica es "mejor" que la otra?, ¿Será que limitar la cantidad de requests es la mejor opción?, ¿Será que por ejemplo, utilizar una cache, es siempre útil? Estas interrogantes serán discutidas y analizadas en detalle más adelante, examinando cada táctica en función de las distintas pruebas y tests que se llevarán a cabo para evaluar su comportamiento.

## 3.4 Descripción de las arquitecturas

### 3.4.1 Caso base

La misma consiste en:

- Artillery: Para simular la carga que va a recibir el sistema.

- Nginx: Actua como reverse proxy. Recibe las solicitudes del cliente y las reenvía al servidor. [\*\*Página oficial\*\*](#).
- Node.js (+ Express): Se encarga de manejar las peticiones de artillery y comunicarse con las APIs externas. [\*\*Página oficial\*\*](#).

Esta arquitectura es la mas sencilla de todas, no tiene ningun componente extra y es contra la que haremos las comparaciones, para discutir si realmente hubo mejoras en la nueva arquitectura, o por el contrario hubo algun atributo de calidad afectado.

### **3.4.2 Caché**

Esta tactica coincide con la base, pero se le agrega:

- Cache de Redis: Con este nuevo componente lo que logramos es que el servidor guarde información en memoria, evitando tener que recurrir a la API externa en múltiples ocasiones. ¿Es siempre util? Lo discutiremos mas adelante. [\*\*Página oficial\*\*](#).

### **3.4.3 Rate Limiting**

Esta tactica coincide con la base, pero se le agrega:

- Rate Limiter (+ Express): En el contexto de Express, el rate limiting es una función que controla la cantidad de solicitudes que un cliente puede hacer a nuestro servidor en un período de tiempo específico. Se puede implementar de forma global, es decir, limitar todos los endpoints a una X cantidad de requests en una Y cantidad de tiempo, o de forma individual personalizada, es decir que cada endpoint responda la cantidad de requests que uno crea necesario en un tiempo que uno crea necesario. Esta ultima es la que puede ajustarse mejor a cada caso. [\*\*Página oficial\*\*](#).

### **3.4.4 Replicación**

Esta tactica coincide con la base, pero se le agrega:

- 3 replicas de nuestro servidor: Nginx puede funcionar como un balanceador de carga (load balancer) para distribuir solicitudes entre múltiples servidores (réplicas) de una aplicación, como una que esté construida con Node.js. Cuando Nginx recibe una solicitud, decide a cuál de las réplicas de la aplicación (servidores Node.js) debe enviarla. Utiliza diferentes algoritmos de balanceo de carga. Por defecto, Nginx utiliza el algoritmo de:
  - **Round Robin:** Las solicitudes se distribuyen de manera equitativa a través de las 3 réplicas.

## **4 Herramientas de mediciones**

Cuando uno se encuentra desarrollando un servicio web, aunque no lo quiera, **debe** saber como se comporta su programa frente a los usuarios. Imaginemos que estamos desarrollando una plataforma de ventas en línea que, en un principio, estaba destinada a atender un volumen moderado de usuarios. El sistema fue diseñado con una arquitectura básica.

La página se lanza y todo parece funcionar bien mientras el tráfico se mantiene dentro de los márgenes esperados. Sin embargo, en un día de promociones especiales, como el "Black Friday", el número de usuarios aumenta drásticamente. Miles de clientes intentan acceder al sitio al mismo tiempo para aprovechar las ofertas. Esto genera una sobrecarga en el servidor, ya que debe procesar una cantidad de solicitudes mucho mayor de lo previsto. Como consecuencia, el servidor web empieza a experimentar latencia elevada, la base de datos se convierte en un cuello de botella, etc. En síntesis ¿en qué se traduce esto? Perdida de dinero. Nuestros clientes, estarán enojados porque no se pudieron efectuar las ventas, y un porcentaje de los usuarios probablemente no vuelva a utilizar nuestro sitio web. Entonces, el lector se preguntará ¿qué hacemos para solucionar esto? Bueno, hoy en día tenemos diferentes herramientas que nos ayudan a analizar cómo se comporta nuestro programa frente a distintos escenarios. ¿Qué escenarios? Un escenario puede ser el que mencionamos anteriormente, que el número de usuarios aumente drásticamente en un intervalo corto de tiempo. Existen otros, y utilizando estas herramientas podemos modelarlos. La herramienta de la cual hablaremos en esta oportunidad es Artillery.

## 4.1 Artillery

### 4.1.1 ¿Qué es y cuál es su objetivo?

Artillery es una herramienta de pruebas de carga (load testing) y rendimiento que se utiliza para medir el rendimiento de aplicaciones y servicios web bajo diferentes niveles de tráfico. Es útil para simular múltiples usuarios concurrentes que realizan solicitudes a un servidor o aplicación, permitiendo identificar cuellos de botella, problemas de latencia y posibles fallos bajo alta carga.

Artillery se destaca por ser fácil de configurar y ejecutar, soportando múltiples protocolos como HTTP, WebSocket y gRPC. Además, es utilizada en el desarrollo de software para garantizar que las aplicaciones puedan escalar y funcionar adecuadamente bajo demanda.

### 4.1.2 ¿Por qué Artillery?

Artillery prioriza la productividad y la satisfacción del desarrollador, y sigue la filosofía de "batteries-included", esto significa que la herramienta ofrece una serie de características y funciones listas para usar desde el principio, facilitando así la experiencia del desarrollador y permitiendo que se enfoque más en las pruebas de rendimiento que en la configuración del entorno.

### 4.1.3 ¿Dónde utilizamos Artillery?

Nuestra aplicación expone los siguientes 4 endpoints:

- **/ping**: Devuelve una respuesta constante y la usaremos como base para las métricas y como healthcheck. No consume ninguna API externa y siempre nos devuelve "Pong".
- **/facts**: Consume la API de Useless Facts. para devolver un hecho aleatorio.
- **/dictionary**: Recurre a la API de Dictionary. para obtener la fonética y definiciones de una palabra que se le pasa por parámetro.

- **/spaceflight\_news:** Obtiene los titulos de las ultimas cinco noticias de la API de Spaceflight News.

Estos 4 endpoints se someterán a pruebas de *carga* y *estres* utilizando artillery, bajo las distintas táticas.

#### 4.1.4 Pruebas de Carga

Las pruebas de carga se enfocan en medir el rendimiento de una aplicación bajo una carga de trabajo esperada o normal. Esto implica simular un número específico de usuarios concurrentes o transacciones durante un período determinado.

#### 4.1.5 Pruebas de Estrés

Las pruebas de estrés implican someter la aplicación a una carga de trabajo más allá de sus capacidades normales, a menudo hasta el punto de fallo. Esto se hace para observar cómo la aplicación se comporta cuando está sobrecargada.

#### 4.1.6 Diferencias Clave

- **Enfoque:** Las pruebas de carga se centran en el rendimiento y la capacidad en situaciones típicas, mientras que las pruebas de estrés se enfocan en la resiliencia y el comportamiento del sistema en situaciones adversas.
- **Objetivos:** Las pruebas de carga buscan verificar el rendimiento y la capacidad, mientras que las pruebas de estrés buscan identificar los límites y la recuperación del sistema.

Ambas pruebas son esenciales para garantizar que una aplicación pueda manejar el tráfico real y se recupere de situaciones adversas, mejorando así la confiabilidad y la experiencia del usuario.

#### 4.1.7 Nuestros escenarios

##### Carga con 4 fases

- **Ramp up:** Se comienza en 2 req/s y se sube progresivamente hasta 8 req/s a lo largo de 30s.
- **Plain:** Se envian 8 req/s durante 30s.
- **Ramp down:** Se desciende de 8 req/s a 2 req/s a lo largo de 30s.
- **Stop:** Se envian 1 req/s durante 30s.

##### Estres con 4 fases

- **Ramp up:** Se comienza en 50 req/s y se sube progresivamente hasta 250 req/s a lo largo de 15s.
- **Plain:** Se envian 300 req/s durante 10s.
- **Ramp down:** Se desciende de 250 req/s a 50 req/s a lo largo de 15s.

- **Stop:** Se envian 1 req/s durante 30s.

Con el objetivo de que se entienda la idea, supongamos que tenemos la tactica "A" y nuestros 4 endpoints.

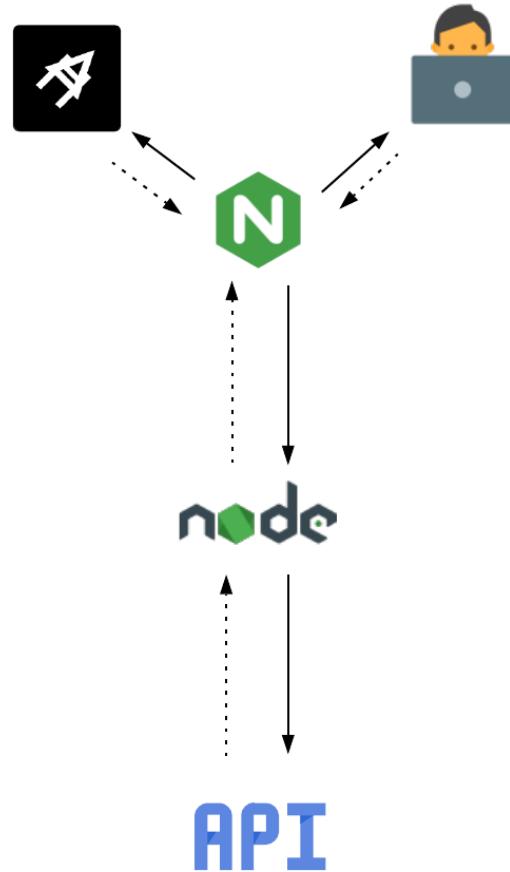


Figure 1: Tactica A.

Comenzamos ejecutando la prueba de carga en cada endpoint bajo la tactica "A", y veremos como se comporta esa arquitectura/tactica frente a este escenario. Por supuesto, antes de cada ejecucion, nuestro deber es preguntarnos ¿Como se comportara la tactica "A", en este endpoint en particular, bajo este escenario? Luego de hacernos las preguntas correspondientes, y de ejecutar nuestro escenario, analizamos...¿tiene sentido lo que estamos observando con respecto a lo que suponiamos? Si suponiamos que la tactica "A", se comportaria de forma aceptable bajo este escenario y endpoint, y los resultados nos dicen que el sistema fallo en reiteradas ocasiones...hay algo que no estamos teniendo en cuenta ¿verdad? Por esta razón, es fundamental realizar pruebas y mediciones que permitan evaluar el comportamiento de cada arquitectura en los distintos escenarios. A continuacion se muestra un ejemplo en codigo de como se estructura un escenario en artillery como el que mencionamos hace un momento.

```

perf > carga > ! space_news.yaml
  1 config:
  2   environments:
  3     api:
  4       target: "http://localhost:5555"
  5     plugins:
  6       statsd:
  7         host: localhost
  8         port: 8125
  9         prefix: "artillery-api"
10
11   pool: 50 # All HTTP requests from all virtual users will be sent over the same connections
12
13   phases:
14     - name: Ramp
15       duration: 30
16       arrivalRate: 2
17       rampTo: 8
18     - name: Plain
19       duration: 30
20       arrivalRate: 8
21     - name: Ramp down
22       duration: 30
23       arrivalRate: 8
24       rampTo: 2
25     - name: Stop
26       duration: 30
27       arrivalRate: 1
28
29
30   scenarios:
31     - name: Space News (/spaceflight_news)
32       flow:
33         - get:
34           url: '/spaceflight_news'

```

Figure 2: Escenario de carga para el endpoint SpaceFlight News

## 4.2 ¿Cómo visualizamos nuestro programa a lo largo del tiempo?

Perfecto, ya hablamos sobre **porque** es necesario realizar mediciones, con que herramienta hacemos estas mediciones y como se ve un escenario de prueba. Ahora bien...¿donde visualizamos estas mediciones? ¿como recopilamos las metricas que artillery nos arroja? Artillery luego de ejecutar una prueba en un determinado endpoint nos arroja metricas, que se ven de la siguiente forma:

```
All VUs finished. Total time: 2 minutes, 3 seconds

-----
Summary report @ 20:40:27(-0300)
-----

http.codes.200: ..... 120
http.request_rate: ..... 1/sec
http.requests: ..... 120
http.response_time:
  min: ..... 1
  max: ..... 1217
  median: ..... 7
  p95: ..... 10.9
  p99: ..... 18
http.responses: ..... 120
vusers.completed: ..... 120
vusers.created: ..... 120
vusers.created_by_name.Space News (/spaceflight_news): ..... 120
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 3
  max: ..... 1263.7
  median: ..... 9.9
  p95: ..... 17.6
  p99: ..... 22.9
```

Figure 3: Resumen de las fases que se ejecutaron en nuestra prueba

Como se puede observar, vemos que en la imagen se visualizan datos muy importantes.

- **Medidas sobre el tiempo de respuesta**
- **Cantidad de requests con exito (code 200)**
- **Cuantos usuarios no pudieron completar su request**
- **Entre otros**

Son datos sumamente importantes, ya que representan el comportamiento de nuestro programa frente a un escenario que decidimos modelar. *No queremos revelar ahora mismo que tactica se utilizo en este caso, de eso hablaremos mas adelante...*

Algo que seria fabuloso, seria ver como se comporto nuestro programa a lo largo del tiempo durante la ejecucion de la prueba. Para lograr eso, artillery envia datos de rendimiento a **Graphite** utilizando el plugin de **StatsD**. StatsD es un demonio que escucha métricas en un formato específico y las reenvía a un backend, como Graphite, para su almacenamiento y visualización. Una vez que Graphite comienza a recibir las metricas, Graphite comienza a enviarselas a **Grafana**. Por supuesto que estas 4 herramientas se deben configurar para que actuen en conjunto, no existe la magia. El objetivo de este informe no es explicar como se configuran, pero si como las utilizamos para poder llegar a nuestras conclusiones y mostrar lo poderosas que son.

A continuacion se muestra un grafico de como se comporta una determinada arquitectura a lo largo del tiempo, desde que comienza la prueba, hasta que finaliza.

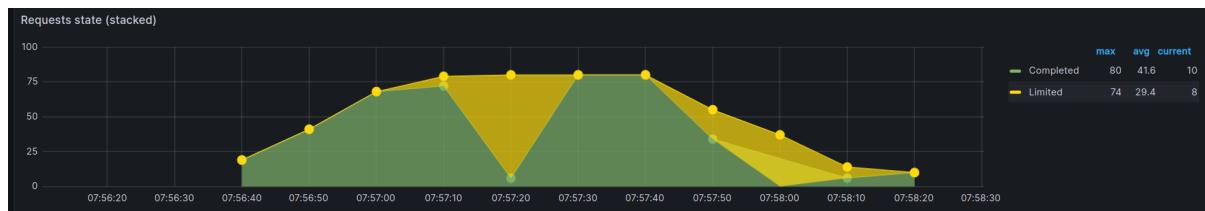


Figure 4: Grafico en grafana

## 5 Ejecución del programa

Por ultimo, en caso de que el lector desee ejecutar nuestro programa debera seguir los siguientes pasos:

1. Debera tener instalado docker ([curso completo de docker](#)).
2. Una vez instalado, se dirigira a [nuestro repositorio en GitHub](#). Alli se clonara nuestro repositorio ([como clonar un repositorio en GitHub](#)).

Nuestro repositorio tiene las siguientes ramas:



3. Supongamos que ya se encuentra en su editor de texto preferido, lo siguiente es moverse a alguna de las ramas.

```
git checkout <branch>
```

4. Ya estando sobre alguna rama, lo siguiente es levantar los contenedores:

```
docker compose up --build
```

Luego de este comando se empezaran a levantar todos los contenedores que incluimos en el archivo *docker-compose.yml*. En esa misma terminal donde ejecuto el comando los podra visualizar.

```
graphite-1 | Applying admin.0003 logentry_add_action_flag_choices... OK
graphite-1 | Applying dashboard.0001 initial... OK
graphite-1 | Applying events.0001 initial... OK
graphite-1 | Applying events.0002 auto 20241003 1651... OK
graphite-1 | Applying sessions.0001 initial... OK
graphite-1 | Applying tagging.0001 initial... OK
graphite-1 | Applying tagging.0002 on_delete... OK
graphite-1 | Applying tagging.0003 adapt_max_tag_length... OK
graphite-1 | Applying tags.0001 initial... OK
graphite-1 | Applying url_shortener.0001 initial... OK
graphite-1 | ok: run: nginx: (pid 60) 4s
graphite-1 | down: brubeck: 1s, normally up, want up
graphite-1 | run: carbon: (pid 86) 5s; run: log: (pid 78) 5s
graphite-1 | run: carbon-aggregator: (pid 65) 5s; run: log: (pid 61) 5s
graphite-1 | down: carbon-relay: 1s, normally up, want up; run: log: (pid 73) 5s
graphite-1 | down: collectd: 1s, normally up, want up
graphite-1 | run: cron: (pid 99) 5s
graphite-1 | down: go-carbon: 1s, normally up, want up
graphite-1 | run: graphite: (pid 63) 5s
graphite-1 | run: nginx: (pid 60) 5s
graphite-1 | down: redis: 1s, normally up, want up
graphite-1 | run: statsd: (pid 66) 5s; run: log: (pid 62) 5s
```

Figure 5: Terminal donde se levantaron los contenedores

5. Deberá abrir otra terminal  **limpia**

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
graphite-1 | run: cron: (pid 59) 5s
graphite-1 | down: go-carbon: 1s, normally up, want up
graphite-1 | run: graphite: (pid 60) 5s
graphite-1 | run: nginx: (pid 62) 5s
graphite-1 | down: redis: 1s, normally up, want up
graphite-1 | run: statsd: (pid 79) 5s; run: log: (pid 76) 5s
grafana-1 | logger=infra.usagestats.collector t=2024-10-03T16:37:11.1954078Z level=info msg="registering usage stat providers" usageStatsProvidersLen=2
grafana-1 | logger=provisioning.alerting t=2024-10-03T16:37:11.198773258Z level=info msg="starting to provision alerting"
grafana-1 | logger=provisioning.alerting t=2024-10-03T16:37:11.198787174Z level=info msg="finished to provision alerting"
grafana-1 | logger=modules t=2024-10-03T16:37:11.200502532Z level=info msg="initialising module=http-server"
grafana-1 | logger=http.server t=2024-10-03T16:37:11.209195379Z level=info msg="HTTP Server Listen" address=[::]:3000 protocol=http subUrl= socket=
grafana-1 | logger=modules t=2024-10-03T16:37:11.210310446Z level=info msg="initialising module=background-services"
grafana-1 | logger=modules t=2024-10-03T16:37:11.210468832Z level=info msg="All modules healthy" modules="[http-server background-services]"
grafana-1 | logger=ngalert.state.manager t=2024-10-03T16:37:11.21510102Z level=info msg="Warming state cache for startup"
grafana-1 | logger=grafanaStorageLogger t=2024-10-03T16:37:11.21517585Z level=info msg="storage starting"
grafana-1 | logger=ngalert.multiorg.alertmanager t=2024-10-03T16:37:11.2155328258Z level=info msg="Starting MultiOrg Alertmanager"
grafana-1 | logger=ngalert.state.manager t=2024-10-03T16:37:11.218450167Z level=info msg="State cache has been initialized" states=0 duration=3.347193ms
grafana-1 | logger=ngalert.scheduler t=2024-10-03T16:37:11.218492076Z level=info msg="Starting scheduler" tickInterval=10s
grafana-1 | logger=ticker t=2024-10-03T16:37:11.2203909668Z level=info msg="starting first tick=2024-10-03T16:37:202
grafana-1 | logger=plugins.update.checker t=2024-10-03T16:37:11.724660094Z level=info msg="Update check succeeded" duration=509.479655ms
grafana-1 | logger=grafana.update.checker t=2024-10-03T16:37:11.731792611Z level=info msg="Update check succeeded" duration=516.665532ms
grafana-1 | ok: run: nginx: (pid 62) 5s

```

Figure 6: Terminal donde se levantaron los contenedores

Figure 7: Terminal limpia

6. En la nueva terminal deberá moverse a la carpeta **/perf** ejecutando:

```
cd perf
```

En esta carpeta podrá visualizar los escenarios de **carga y estres (stress)**

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
tincho@tincho-System-Product-Name:~/Documents/Facultad/Arquitectura del Software/RepositorioTP1/2c24-tp1-arquitectura-de-software/perf$ ls
carga dashboard.json datasets node_modules package.json root.yaml run-scenario.sh stress
tincho@tincho-System-Product-Name:~/Documents/Facultad/Arquitectura del Software/RepositorioTP1/2c24-tp1-arquitectura-de-software/perf$ 

```

Figure 8: Documentos en la carpeta /perf

7. En este momento deberemos instalar la herramienta de prueba de carga **Artillery** y **npm** para poder ejecutar nuestras pruebas.

```
sudo apt install npm
```

```
sudo npm install -g artillery
```

8. Ahora nos resta elegir que *endpoint* queremos poner a prueba y que *escenario* vamos a correr.

```
✓ carga
! dictionary.yaml
! facts.yaml
! ping.yaml
! space.yaml
> datasets
> node_modules
✓ stress
! dictionary.yaml
! facts.yaml
! ping.yaml
! space.yaml
```

Figure 9: Endpoints a los cuales podemos ejecutarles las pruebas

```
npm run scenario <escenario>/<endpoint> api
```

Si quisieramos poner a prueba el endpoint "space" bajo el escenario "carga", entonces el comando se veria asi:

```
npm run scenario carga/space api
```

9. Esperar a que se ejecute la prueba y analizar las metricas que artillery nos arroja por la terminal :)

## 6 Nuestras mediciones

Ya comentamos las herramientas y tacticas que se utilizaron en este trabajo practico. Ahora es tiempo de debatir nuestras mediciones.

Para que el lector lo tenga presente, en la sección 4.1.3 se realizo una descripcion general de los endpoints que tiene nuestra aplicacion. Leyendo en detalle, quizas, podria anticipar algunas de nuestras mediciones. ¿Porque? Por ejemplo, observando la descripcion del endpoint `/ping`, vemos que siempre devuelve un valor constante sin consumir una API externa, en comparacion a los otros endpoints que si consumen una API externa...

### 6.1 Caso base

#### 6.1.1 Escenario: Carga , Endpoint: /ping

El consumo de recursos se mantuvo bajo, tal como se anticipaba. El tiempo de respuesta fue considerablemente reducido, lo cual resulta coherente, dado que este endpoint no realiza consultas ni procesamiento significativos. Todas las 570 solicitudes fueron procesadas exitosamente.

El escenario de carga muestra claramente cada una de sus fases. En la fase de ramp-up, se inició con 2 solicitudes por segundo, aumentando progresivamente hasta alcanzar 8 solicitudes por segundo. Posteriormente, en la fase de estabilización (Plain), la carga se mantuvo constante en 8 solicitudes por segundo. Luego, en la fase de ramp-down, el sistema continuó con 8 solicitudes por segundo antes de disminuir gradualmente hasta llegar a 2 solicitudes por segundo. Finalmente, en la fase de detención (Stop), se observa un descenso a 1 solicitud por segundo.

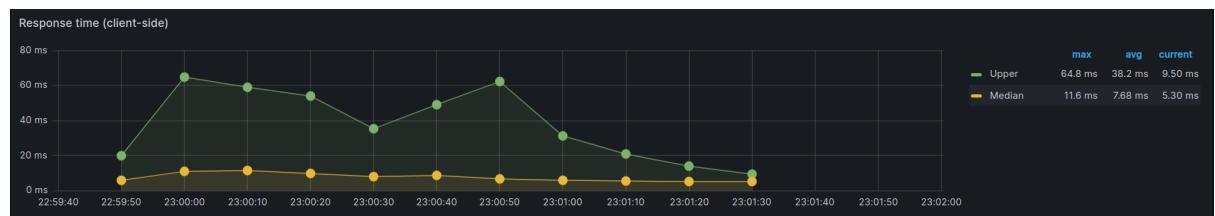


Figure 10: Caso base - Carga - Ping

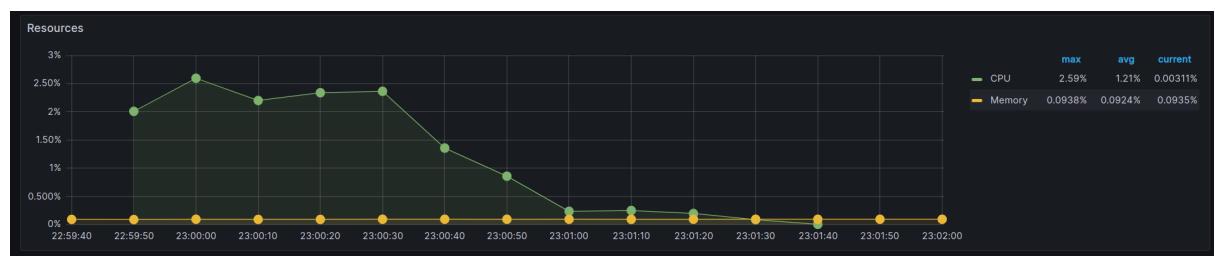


Figure 11: Caso base - Carga - Ping



Figure 12: Caso base - Carga - Ping

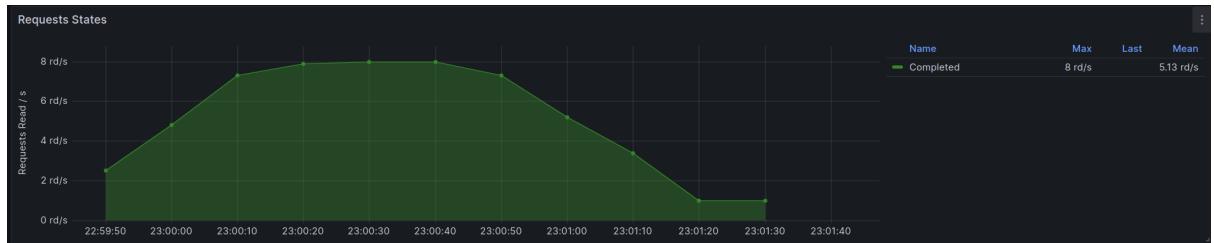


Figure 13: Caso base - Carga - Ping

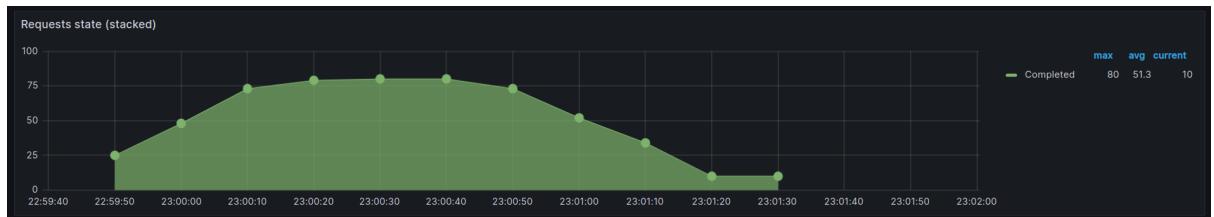


Figure 14: Caso base - Carga - Ping



Figure 15: Caso base - Carga - Ping

### 6.1.2 Escenario: Carga , Endpoint: /facts

No se observa prácticamente ninguna diferencia entre el tiempo de respuesta total y el tiempo de respuesta de la API externa, lo cual es lógico, ya que la respuesta obtenida requiere un procesamiento mínimo. Por lo tanto, se puede concluir que el tiempo de respuesta está directamente asociado a la API externa.

El consumo de recursos mostró un incremento en comparación con el endpoint de ping. Asimismo, se observó una disminución en el uso de recursos durante la fase de ramp-down. Las 80 solicitudes fueron procesadas correctamente.

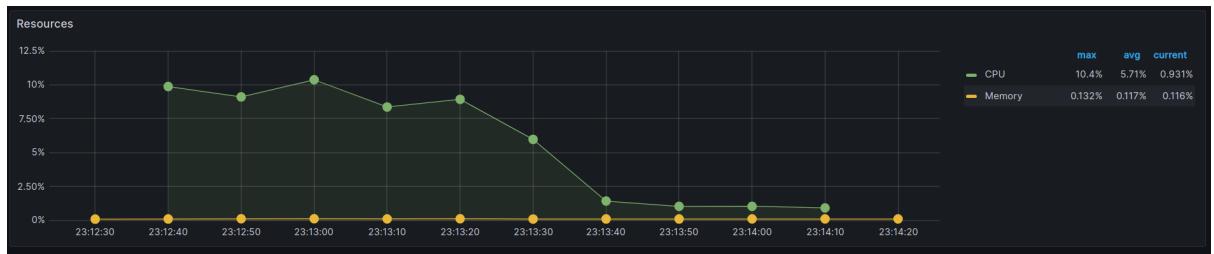


Figure 16: Caso base - Carga - Facts



Figure 17: Caso base - Carga - Facts

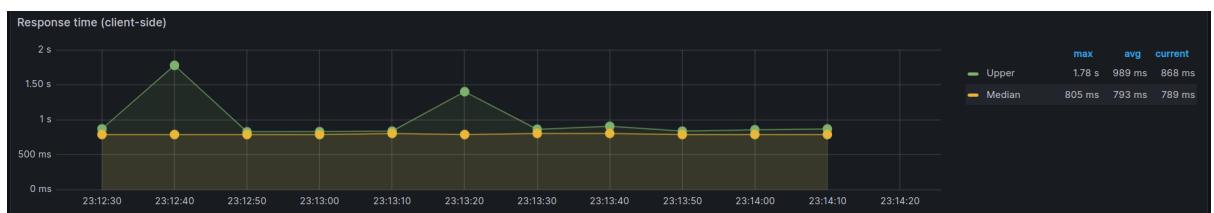


Figure 18: Caso base - Carga - Facts

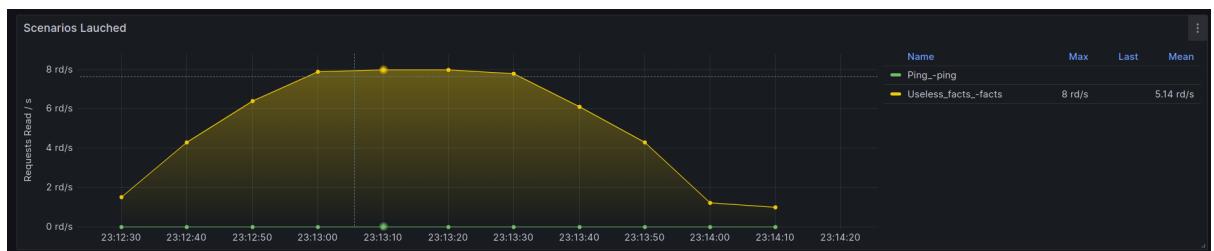


Figure 19: Caso base - Carga - Facts

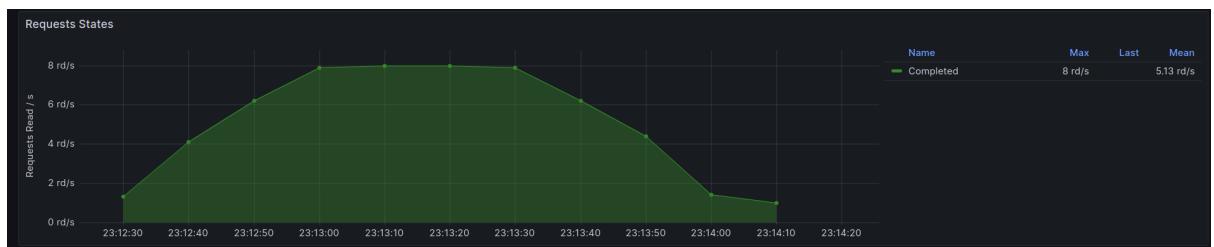


Figure 20: Caso base - Carga - Facts



Figure 21: Caso base - Carga - Facts

### 6.1.3 Escenario: Carga , Endpoint: /dictionary

No se perciben diferencias entre el tiempo de respuesta total y el de la API externa, lo cual es lógico, dado que la respuesta obtenida requiere un procesamiento mínimo. En consecuencia, se puede afirmar que el tiempo de respuesta corresponde principalmente a la API externa.

El uso de recursos incrementa significativamente, lo cual es comprensible dado que los resultados de la API de "dictionary" son más extensos que los de "useless facts", lo que implica que la respuesta enviada al cliente también es más grande.

De las 80 solicitudes realizadas, solo se completó la mitad. Las restantes no se procesaron debido a limitaciones de tiempo.

Además, se observó que con esta arquitectura, el sistema puede manejar un máximo de 3 a 4 solicitudes por segundo. Al exceder este límite, las solicitudes se restringen, pero cuando se manejan 3 o menos solicitudes por segundo, el porcentaje de respuestas exitosas es cercano al 100%. Dado que en este caso no se implementó ninguna limitación propia, se concluye que la API de "dictionary" tiene un límite de solicitudes integrado.

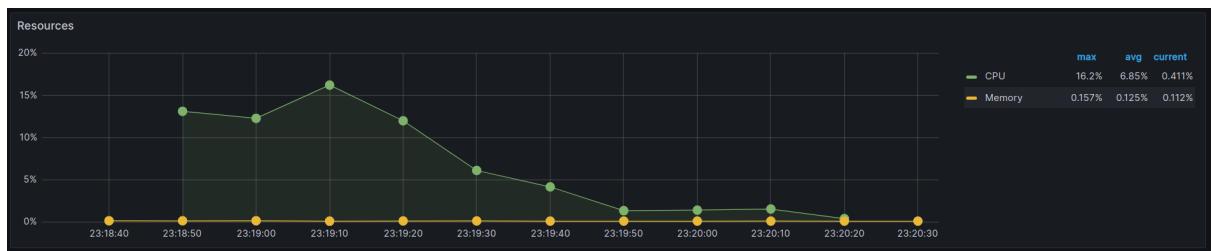


Figure 22: Caso base - Carga - Dictionary

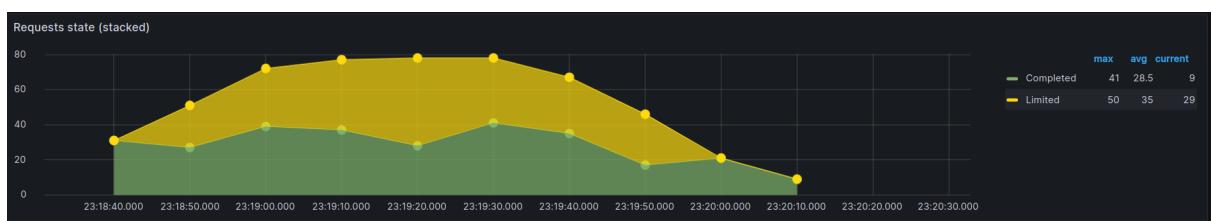


Figure 23: Caso base - Carga - Dictionary

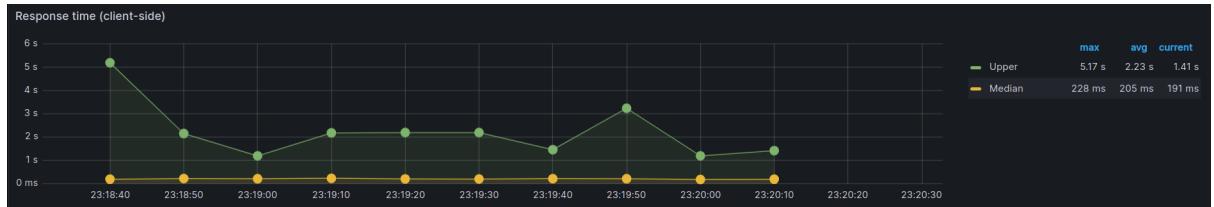


Figure 24: Caso base - Carga - Dictionary

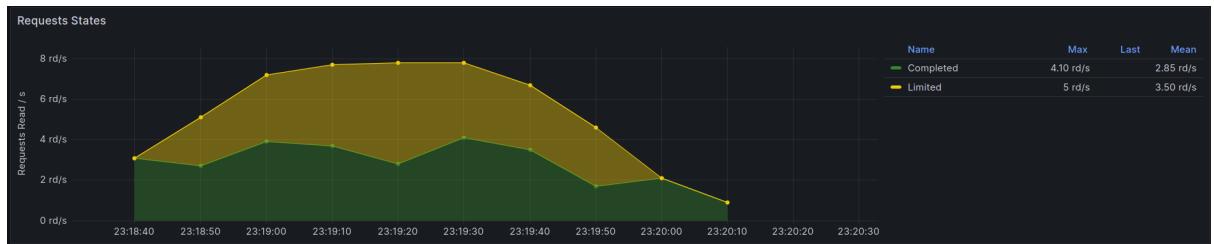


Figure 25: Caso base - Carga - Dictionary



Figure 26: Caso base - Carga - Dictionary



Figure 27: Caso base - Carga - Dictionary

#### 6.1.4 Escenario: Carga , Endpoint: /spaceflight\_news

Se observa un aumento significativo en el tiempo de respuesta y el consumo de recursos en relación al endpoint ping. Este incremento se debe a la necesidad de obtener información de una API externa, lo cual queda evidenciado por la presencia del parámetro external time, que no estaba presente en ejecuciones anteriores.

A pesar de este aumento en los tiempos de procesamiento, todas las solicitudes fueron respondidas con éxito

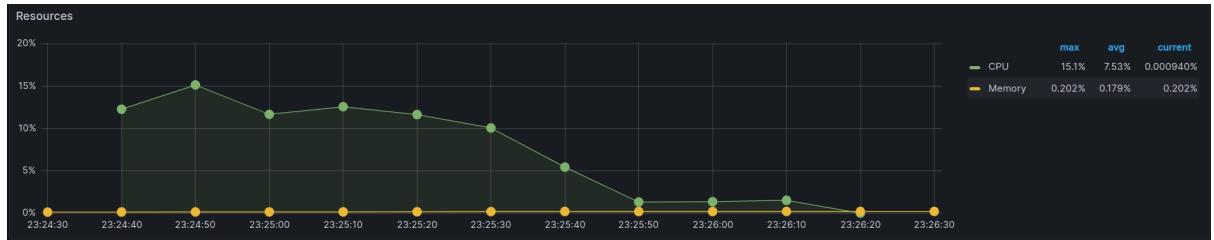


Figure 28: Caso base - Carga - SpaceFlight News

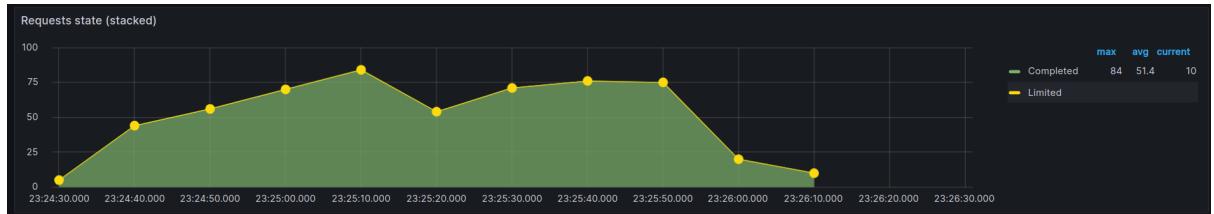


Figure 29: Caso base - Carga - SpaceFlight News



Figure 30: Caso base - Carga - SpaceFlight News

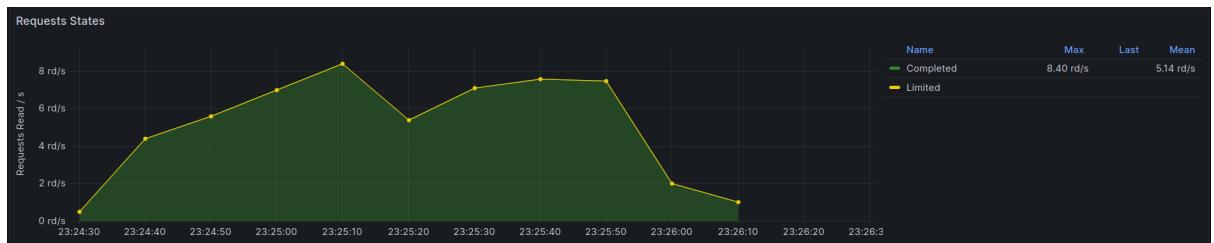


Figure 31: Caso base - Carga - SpaceFlight News



Figure 32: Caso base - Carga - SpaceFlight News

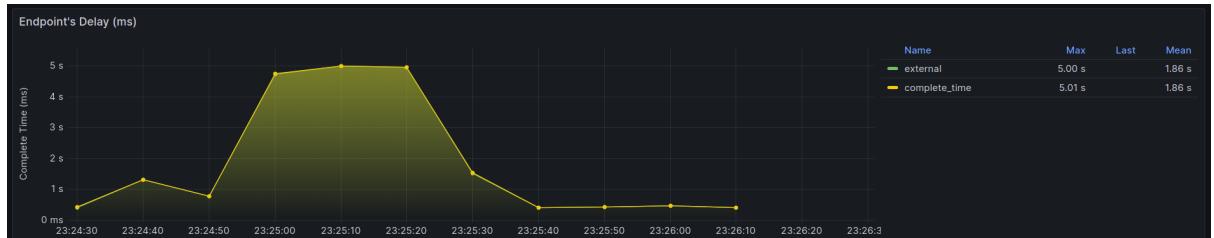


Figure 33: Caso base - Carga - SpaceFlight News

### 6.1.5 Escenario: Estres , Endpoint: /ping

El tiempo de respuesta prácticamente no mostró un incremento significativo en comparación con los resultados obtenidos en el escenario anterior.

Aunque se observaron algunos picos máximos, el gráfico se actualiza cada 10 segundos, por lo que no toda la información se refleja de manera precisa en el mismo. Al observar el promedio, dichos picos máximos parecen ser outliers.

Se notó un aumento considerable en el uso de recursos, comenzando con un consumo bajo tanto al inicio como al final del proceso.

A pesar de tratarse de un endpoint básico y de fácil procesamiento, la gran cantidad de solicitudes generadas provocó algunos fallos durante la fase de plain, en la que se gestionaron el mayor número de solicitudes por segundo.

El escenario consta de cuatro etapas bien definidas. La primera es una fase de ramp-up, comenzando con 50 solicitudes por segundo y alcanzando progresivamente 250 solicitudes por segundo. A continuación, se desarrolla la fase plain, en la cual se procesan 300 solicitudes por segundo. Posteriormente, en la fase de ramp-down, el número de solicitudes disminuye gradualmente de 250 a 50 solicitudes por segundo. Finalmente, se entra en la fase de stop, con una sola solicitud por segundo.

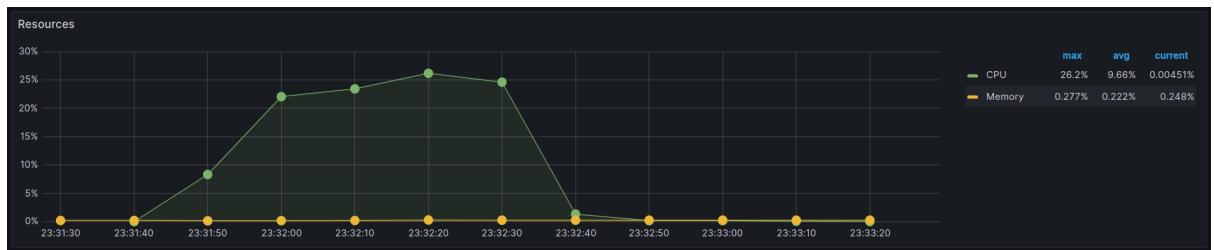


Figure 34: Caso base - Estres - Ping



Figure 35: Caso base - Estres - Ping

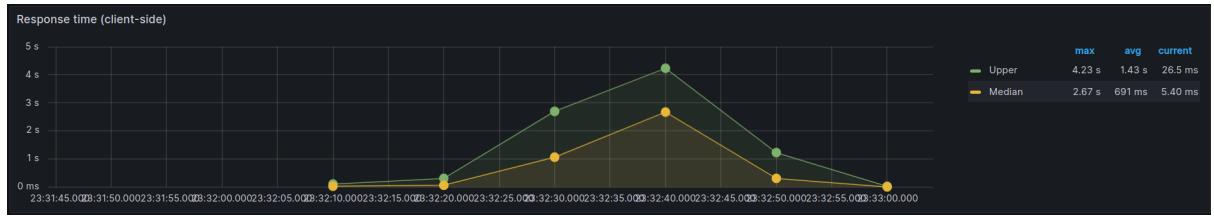


Figure 36: Caso base - Estres - Ping



Figure 37: Caso base - Estres - Ping



Figure 38: Caso base - Estres - Ping

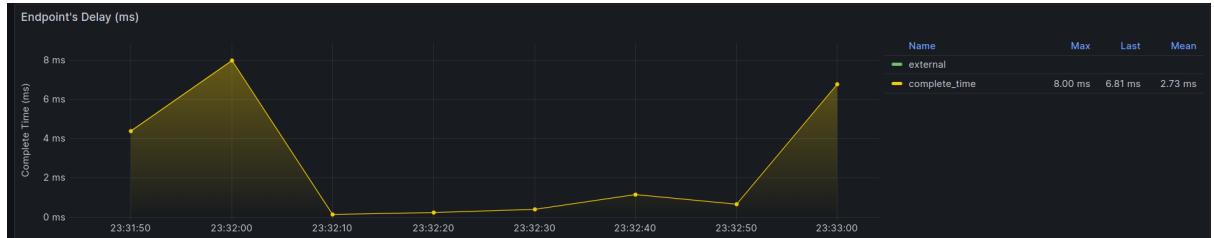


Figure 39: Caso base - Estres - Ping

### 6.1.6 Escenario: Estres , Endpoint: /facts

Se concluye que la arquitectura actual no es capaz de soportar un tráfico más elevado, ya que la mayoría de las solicitudes fallaron debido a una demanda excesiva. Esto se evidencia en el hecho de que las peticiones a la API externa experimentaron un retraso considerable en comparación con el escenario de carga inicial, lo que generó una mayor latencia en nuestra aplicación. Eventualmente, el sistema dejó de responder.

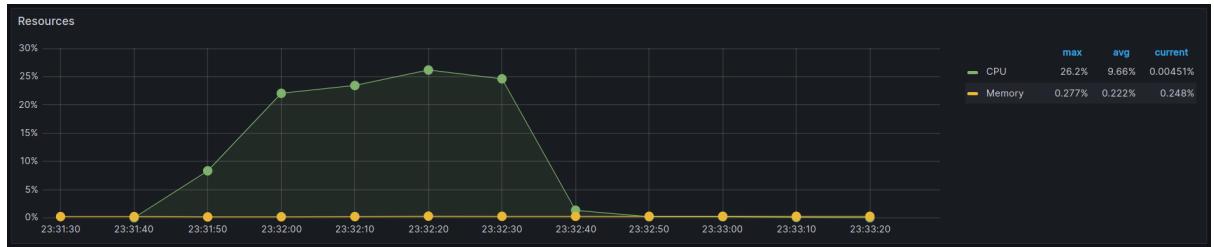


Figure 40: Caso base - Estres - Facts



Figure 41: Caso base - Estres - Facts



Figure 42: Caso base - Estres - Facts

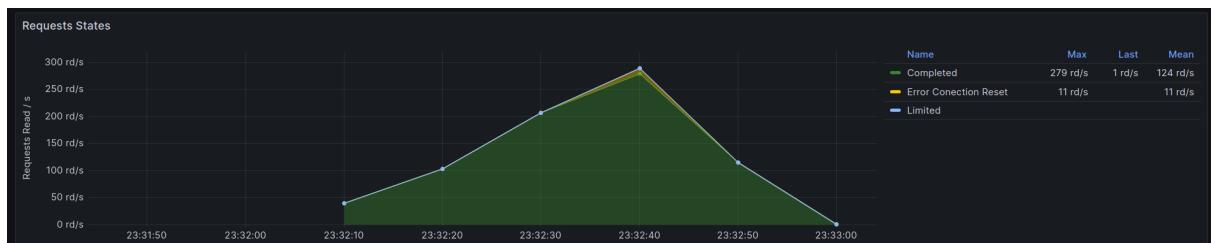
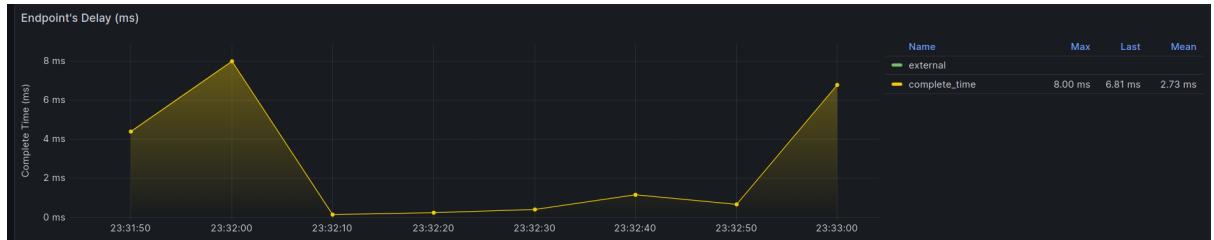


Figure 43: Caso base - Estres - Facts



Figure 44: Caso base - Estres - Facts



### 6.1.7 Escenario: Estres , Endpoint: /dictionary

Se observa que, a medida que la demanda aumenta, el tiempo de respuesta también incrementa, hasta que eventualmente el sistema deja de responder, ya que permanece esperando las respuestas de las primeras solicitudes.

El tiempo de respuesta se incrementa considerablemente en comparación con el escenario de carga anterior.

Se aprecia un aumento significativo en el uso de recursos. No obstante, dicho uso disminuye abruptamente cuando se deja de procesar solicitudes debido a la alta demanda generada, lo que provoca que no se gestionen las nuevas solicitudes recibidas y, en consecuencia, el consumo de CPU se reduce prácticamente a cero.

El porcentaje de respuestas es bajo, ya que solo se gestionaron las primeras solicitudes; a medida que transcurre el tiempo, el sistema deja de responder por completo.



Figure 46: Caso base - Estres - Dictionary



Figure 47: Caso base - Estres - Dictionary

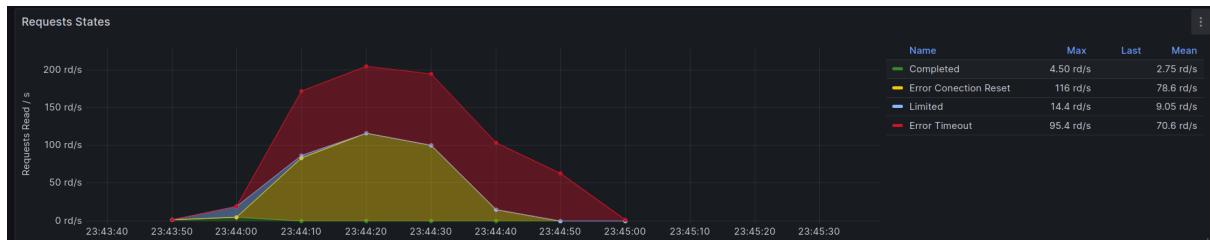


Figure 48: Caso base - Estres - Dictionary



Figure 49: Caso base - Estres - Dictionary



Figure 50: Caso base - Estres - Dictionary



Figure 51: Caso base - Estres - Dictionary

### 6.1.8 Escenario: Estres , Endpoint: /spaceflight\_news

Se ha registrado un incremento de casi el doble en el tiempo requerido para proporcionar una respuesta en comparación con el escenario de carga anterior.

El uso de recursos se duplicó con respecto al escenario de carga anterior.

Se observó una mayor proporción de fallos en comparación con las respuestas exitosas obtenidas.



Figure 52: Caso base - Estres - SpaceFlight News



Figure 53: Caso base - Estres - SpaceFlight News



Figure 54: Caso base - Estres - SpaceFlight News

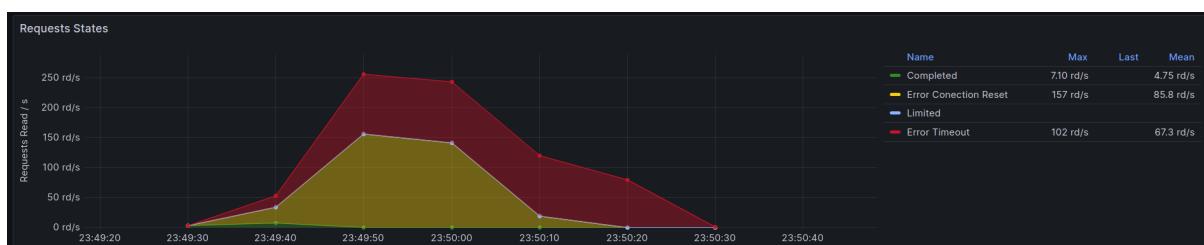


Figure 55: Caso base - Estres - SpaceFlight News

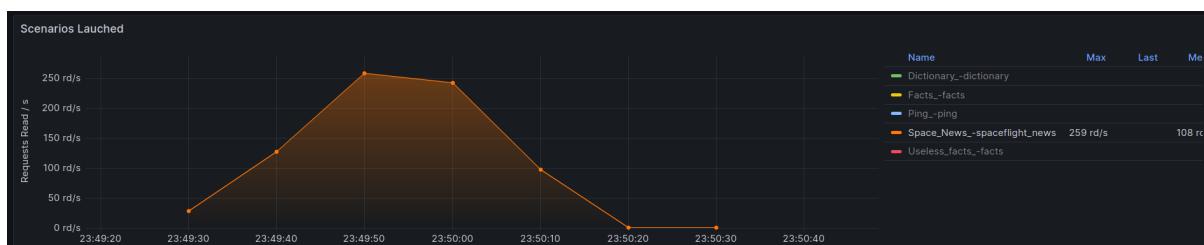


Figure 56: Caso base - Estres - SpaceFlight News



Figure 57: Caso base - Estres - SpaceFlight News

## 6.2 Caché

### 6.2.1 Escenario: Carga , Endpoint: /ping

No se observan diferencias en esta prueba con respecto al caso base, ya que no se implementó la caché para este endpoint. La razón de no haber implementado la caché radica en que, en el escenario original, el endpoint no realizaba consultas a APIs externas, por lo que no había información relevante para almacenar.

### 6.2.2 Escenario: Carga , Endpoint: /facts

No se observan diferencias en esta prueba con respecto al caso base, ya que no se implementó la caché para este endpoint. Aunque este endpoint realiza consultas a una API externa, la información obtenida en cada solicitud es de carácter aleatorio. Por este motivo, no tiene sentido utilizar la caché, dado que su propósito es almacenar datos repetitivos, y en este caso se busca que la información sea aleatoria y no recurrente.

### 6.2.3 Escenario: Carga , Endpoint: /dictionary

Optamos por implementar la caché en este endpoint debido a que, aunque la probabilidad de repetición de las palabras solicitadas no es alta, existe la posibilidad de que algunas se repitan. Estas palabras se almacenarán en la caché por un periodo de tiempo reducido.

Se observó una reducción significativa en el tiempo de respuesta, atribuible al hecho de que, al contar con información almacenada en la caché susceptible de repetirse, se realizan menos consultas a la API externa. Este mismo factor también contribuyó a una disminución en el consumo de recursos.

En comparación con el caso base, se detecta un aumento en la cantidad de solicitudes procesadas por segundo. Aunque en un principio se concluyó que la API externa era un factor limitante en cuanto a la cantidad de requests por segundo, es probable que esta limitación se haya reducido al aprovechar la caché.



Figure 58: Cache - Carga - Dictionary



Figure 59: Cache - Carga - Dictionary

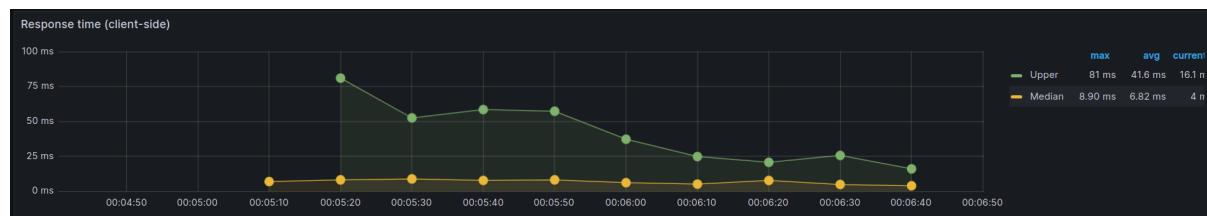


Figure 60: Cache - Carga - Dictionary



Figure 61: Cache - Carga - Dictionary

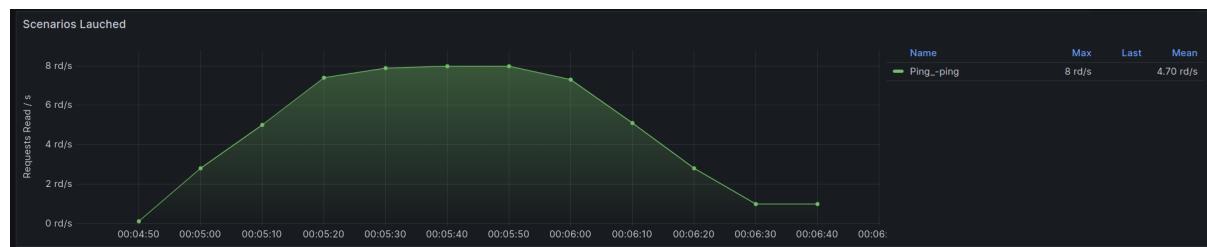


Figure 62: Cache - Carga - Dictionary

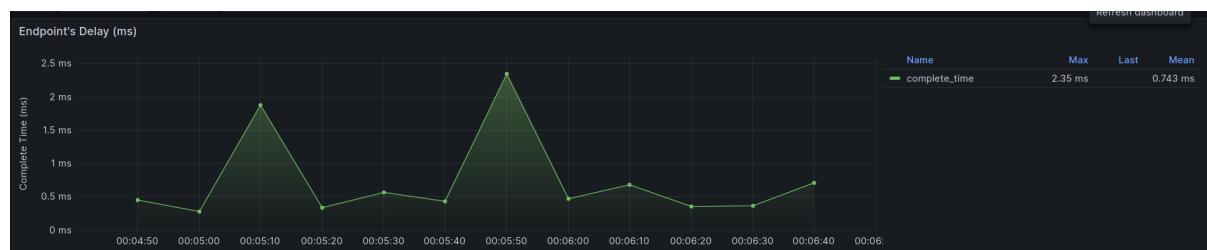


Figure 63: Cache - Carga - Dictionary

#### 6.2.4 Escenario: Carga , Endpoint: /spaceflight\_news

Consideramos prudente almacenar la información en la caché por un lapso de 1 minuto, aunque esto representa solo el 10% de la frecuencia con la que se actualizan los datos, según la información proporcionada por la API. Esta decisión se tomó para reducir considerablemente la cantidad de solicitudes a la API externa, evitando al mismo tiempo el riesgo de proporcionar información desactualizada durante un periodo prolongado.

Gracias a esta implementación, observamos que solo se realizan dos consultas a la API externa, y el tiempo de respuesta se reduce de manera abrupta. Por esta misma razón, también se registra una disminución significativa en el consumo de recursos.



Figure 64: Cache - Carga - Spaceflight News



Figure 65: Cache - Carga - Spaceflight News



Figure 66: Cache - Carga - Spaceflight News

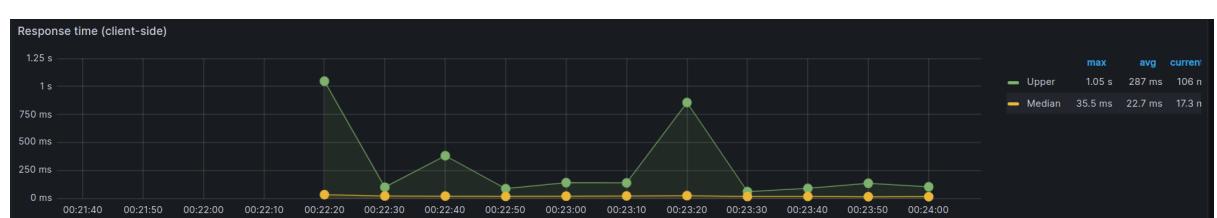


Figure 67: Cache - Carga - Spaceflight News

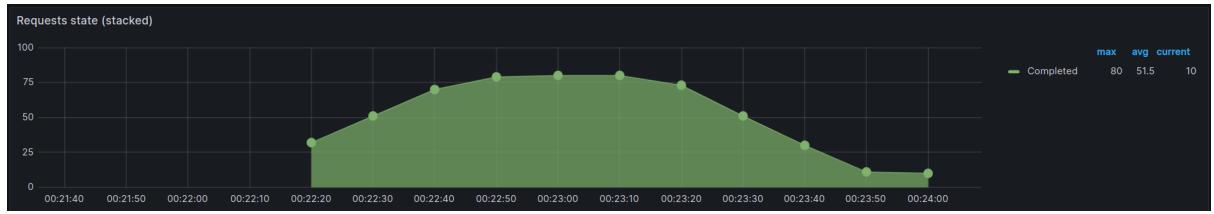


Figure 68: Cache - Carga - Spaceflight News

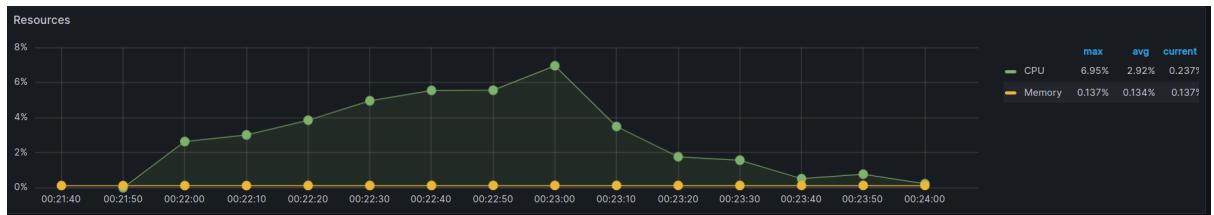


Figure 69: Cache - Carga - Spaceflight News

### 6.2.5 Escenario: Estres , Endpoint: /ping

No se observan diferencias entre esta prueba y la que se realizó en el caso base, dado que no se ha implementado la caché para este endpoint.

### 6.2.6 Escenario: Estres , Endpoint: /facts

No se observan diferencias entre esta prueba y la que se realizó en el caso base, dado que no se ha implementado la caché para este endpoint. ¿Qué hubiese sucedido si hubiésemos utilizado la cache en este endpoint en particular?

- Utilizar caché para datos que no necesitan ser cacheados puede malgastar memoria y recursos del sistema. Como las cachés suelen almacenarse en RAM (un recurso costoso), esto puede llevar a un uso inefficiente de la memoria, especialmente si los datos almacenados no son consultados con frecuencia.

### 6.2.7 Escenario: Estres , Endpoint: /dictionary

Aunque se emplea la caché para este endpoint, debido a la baja probabilidad de repetición de palabras, con un gran volumen de solicitudes se alcanza el límite de memoria, lo que vuelve a generar el cuello de botella observado en el caso base. Ya que si las palabras no fueron almacenadas en la cache, no hay diferencia con el caso base, se comportan de la misma manera al recibir la siguiente request.

Estas interrupciones fueron provocadas por el alto volumen de solicitudes, lo que causó un retraso en el procesamiento del endpoint.

El mayor consumo de memoria se debe al uso de la caché, tanto para almacenar como para recuperar información.

Sin embargo, se logró reducir la cantidad de fallos en comparación con el caso base.



Figure 70: Cache - Estres - Dictionary

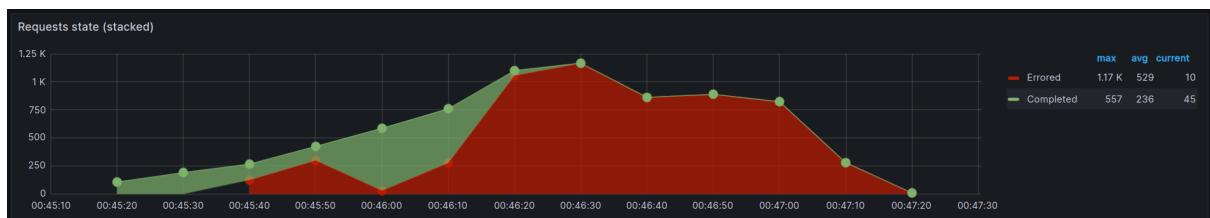


Figure 71: Cache - Estres - Dictionary

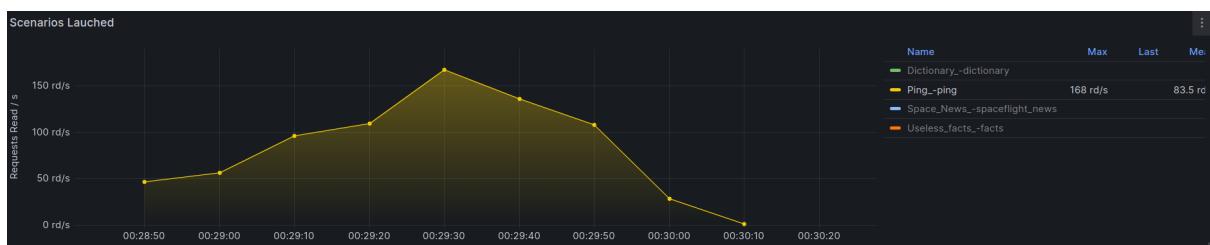


Figure 72: Cache - Estres - Dictionary

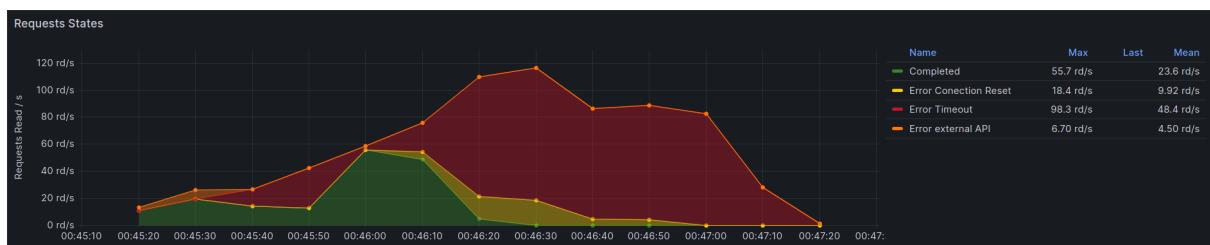


Figure 73: Cache - Estres - Dictionary



Figure 74: Cache - Estres - Dictionary

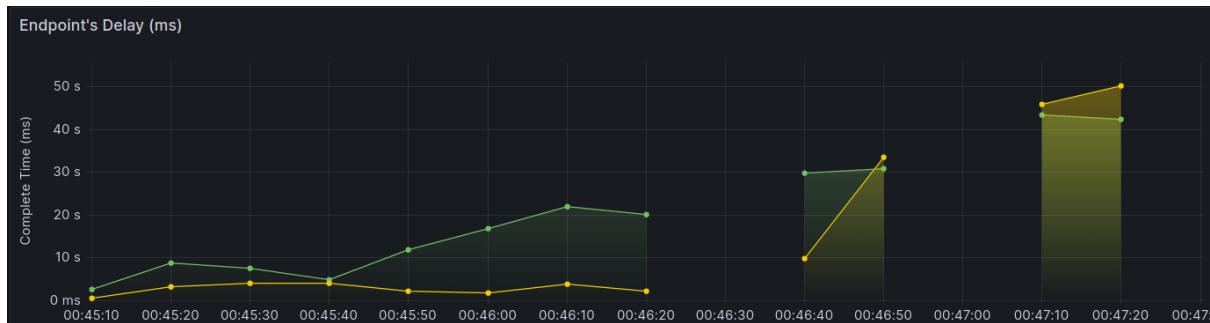


Figure 75: Cache - Estres - Dictionary

```
node-1 { error: 'Error al obtener la palabra del diccionario.' } Request failed with status code 429
redis-1 1:M 01 Oct 2024 03:50:35.075 * 100 changes in 300 seconds. Saving...
redis-1 1:M 01 Oct 2024 03:50:35.077 * Background saving started by pid 22
redis-1 22:C 01 Oct 2024 03:50:35.128 * DB saved on disk
redis-1 22:C 01 Oct 2024 03:50:35.135 * Fork CoW for RDB: current 0 MB, peak 0 MB, average 0 MB
redis-1 1:M 01 Oct 2024 03:50:35.178 * Background saving terminated with success
node-1 { error: 'Error al obtener la palabra del diccionario.' } Request failed with status code 429
node-1 { error: 'Error al obtener la palabra del diccionario.' } Request failed with status code 429
```

Figure 76: Cache - Estres - Dictionary

### 6.2.8 Escenario: Estres , Endpoint: /spaceflight\_news

Para este endpoint, se ha observado una mejora significativa debido a la implementación de la caché, lo que reduce al mínimo las peticiones a la API externa y, en consecuencia, mejora el rendimiento.

En el `external_time`, se aprecia un lapso entre la expiración de una clave en la caché y el almacenamiento de una nueva entrada, durante el cual se realizan múltiples solicitudes nuevas. Como la información aún no se encuentra en la caché, estas solicitudes también consultan la API externa hasta que una de ellas obtiene una respuesta. Sin embargo, todas las solicitudes realizadas durante este periodo esperarán su respectiva respuesta y no harán uso de la caché.

Aunque se generan menos consultas a la API externa, el uso de la caché provoca que el consumo de recursos se mantenga en niveles similares.

A diferencia del caso base, se observa un alto porcentaje de solicitudes respondidas con éxito, cercano al total.



Figure 77: Cache - Estres - SpaceFlight News



Figure 78: Cache - Estres - SpaceFlight News

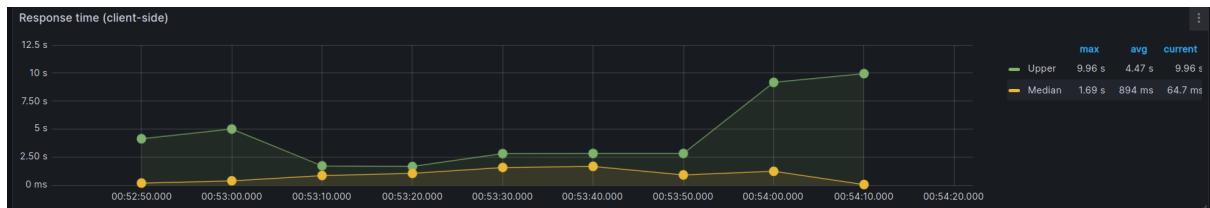


Figure 79: Cache - Estres - SpaceFlight News



Figure 80: Cache - Estres - SpaceFlight News



Figure 81: Cache - Estres - SpaceFlight News

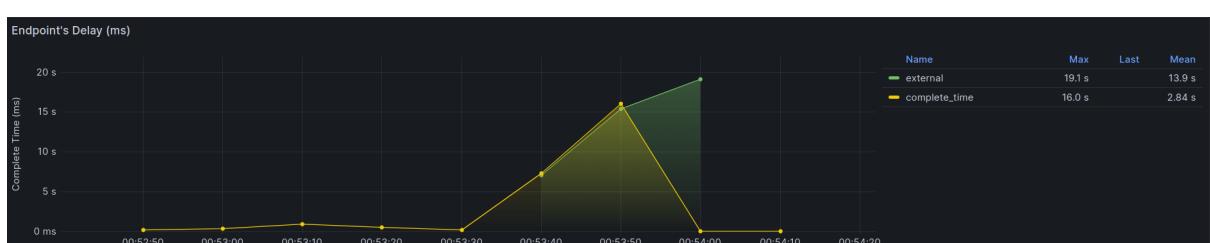


Figure 82: Cache - Estres - SpaceFlight News

## 6.3 Rate Limiting

En términos generales, se observa que esta táctica no presenta mejoras significativas en comparación con el caso base.

No obstante, sí mejora la disponibilidad, ya que limita la cantidad de solicitudes que un mismo usuario puede realizar en un tiempo determinado. Esto contribuye a evitar la caída de la API. Como consecuencia de la reducción de la carga, se logra una disminución en el consumo de recursos.

Este enfoque es particularmente conveniente en un escenario donde existen múltiples clientes, ya que, si uno de ellos está saturando el sistema, esta metodología incrementa las posibilidades de brindar respuestas a un mayor número de clientes. A su vez, es muy conveniente para la seguridad de nuestro sistema. Imaginemos que estamos recibiendo un ataque de un cliente el cual quiere lograr que nuestro sistema caiga enviando miles de requests por segundo, si nuestro sistema no pudiese controlar esa cantidad de requests el resultado sería nuestro sistema caido, pero, gracias a que incluimos este componente en nuestra arquitectura logramos que ese ataque no se lleve a cabo, debido a que lo estamos *limitando*.

### 6.3.1 Escenario: Carga , Endpoint: /ping

En este escenario no se observó una mejora con la táctica aplicada en comparación con el caso base. Esto se debe a que, en el caso base, este escenario se procesaba sin problemas. Sin embargo, al implementar una limitación general en la cantidad de solicitudes por segundo permitidas por una IP, se alcanzó dicho límite, lo que resultó en una menor cantidad de respuestas exitosas.

Se observa claramente que el límite se reinicia cada 45 segundos, tras lo cual se aceptan nuevas peticiones. Por esta razón, hay un intervalo en el que no se registran respuestas, ya que todas las solicitudes recibidas durante ese periodo fueron rechazadas debido a que se había alcanzado el límite.

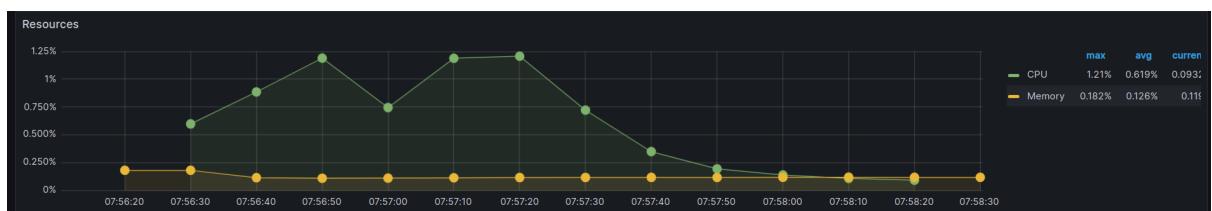


Figure 83: Rate Limiting - Carga - Ping



Figure 84: Rate Limiting - Carga - Ping

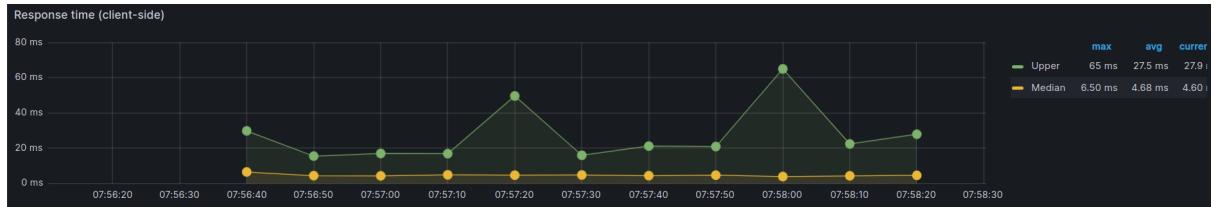


Figure 85: Rate Limiting - Carga - Ping

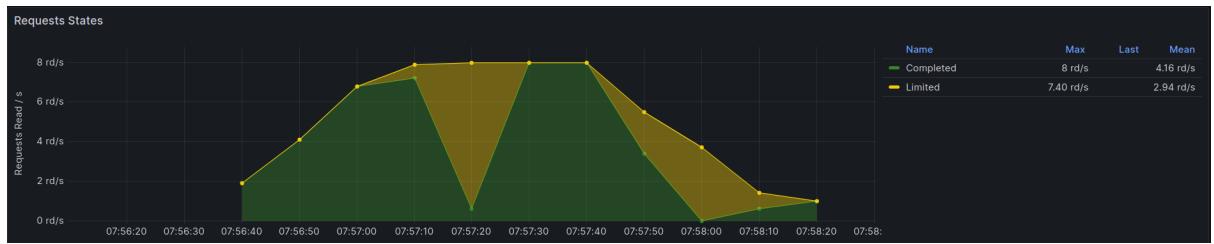


Figure 86: Rate Limiting - Carga - Ping



Figure 87: Rate Limiting - Carga - Ping



Figure 88: Rate Limiting - Carga - Ping

### 6.3.2 Escenario: Carga , Endpoint: /facts

En este escenario, no se observó una mejora con la táctica aplicada en comparación con el caso base. Esto se debe a que, en el caso base, este escenario se procesaba sin problemas. Sin embargo, al implementar una limitación general en la cantidad de solicitudes por segundo permitidas por una IP, se alcanzó rápidamente dicho límite, lo que resultó en una menor cantidad de respuestas exitosas.

Es evidente que, al establecer un límite que se reinicia cada 45 segundos, el sistema vuelve a aceptar nuevas peticiones una vez transcurrido ese tiempo. Esto provoca un periodo en el que no se registran respuestas, ya que todas las solicitudes recibidas durante ese intervalo fueron rechazadas por haber alcanzado el límite.

Sin embargo, esta técnica, al limitar la cantidad de solicitudes, permitió reducir el consumo de recursos.

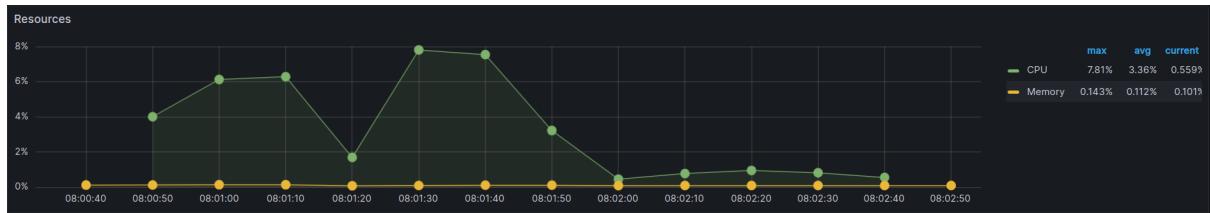


Figure 89: Rate Limiting - Carga - Facts



Figure 90: Rate Limiting - Carga - Facts

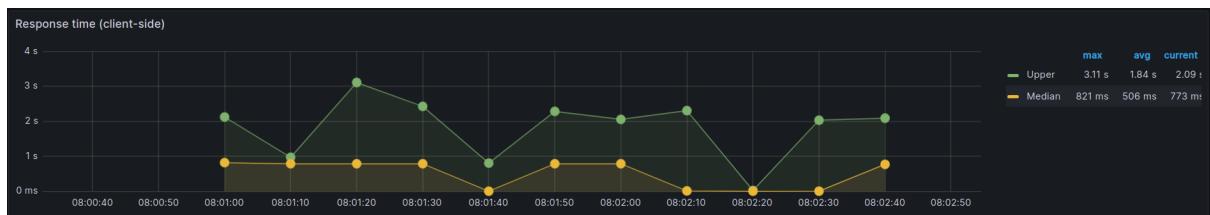


Figure 91: Rate Limiting - Carga - Facts

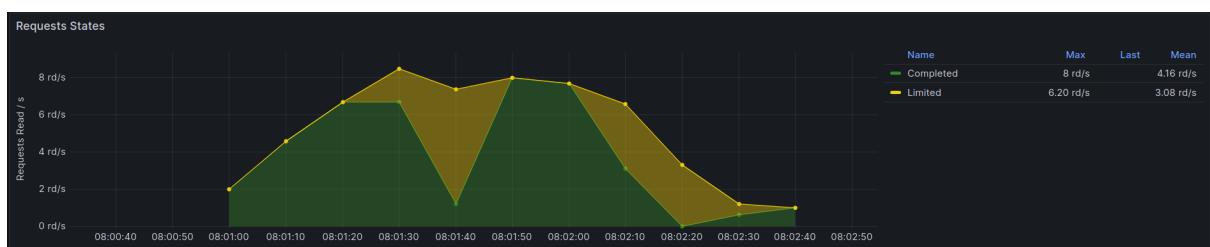


Figure 92: Rate Limiting - Carga - Facts

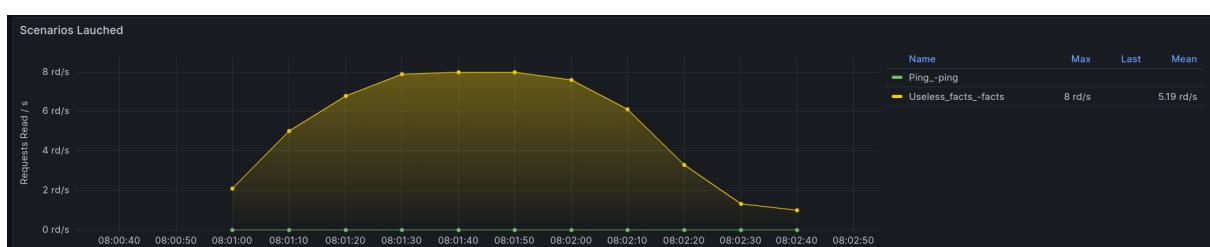


Figure 93: Rate Limiting - Carga - Facts

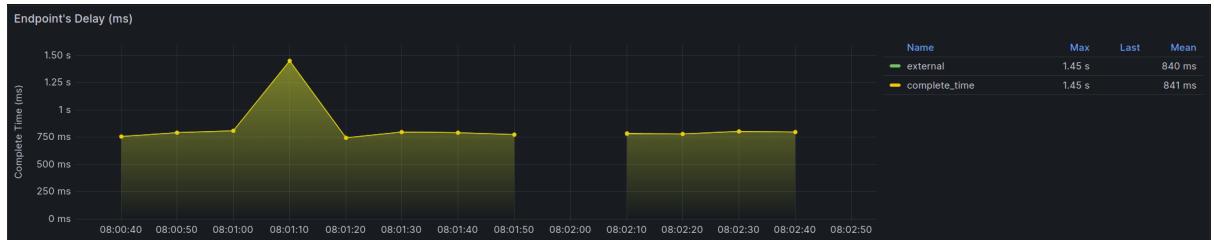


Figure 94: Rate Limiting - Carga - Facts

### 6.3.3 Escenario: Carga , Endpoint: /dictionary

Considerando la hipótesis de que la API del diccionario tiene una tolerancia de 4 solicitudes por segundo, este escenario tuvo un peor rendimiento en comparación con el caso base. Esto se debe a que ahora se enfrenta la limitación de ambas APIs.

Aunque nuestra limitación permite un mayor número de solicitudes, opera en un rango de tiempo más amplio, lo que genera una menor tolerancia en comparación con la API externa. Al combinar ambas restricciones, el resultado es que se procesan menos solicitudes que en el caso base.

Sin embargo, debido a esta situación, también se percibe una mejora en la utilización de los recursos.



Figure 95: Rate Limiting - Carga - Dictionary



Figure 96: Rate Limiting - Carga - Dictionary

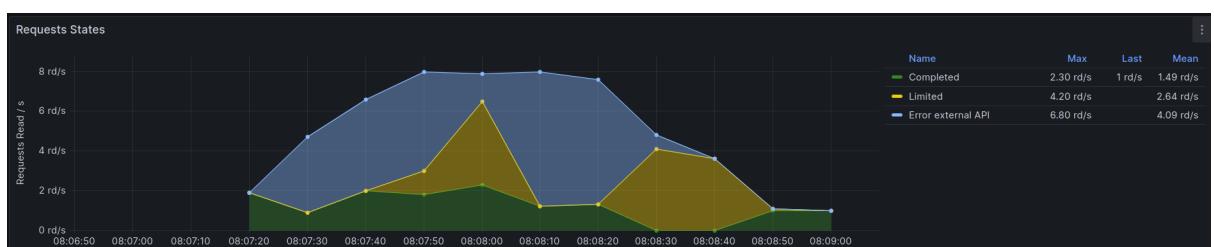


Figure 97: Rate Limiting - Carga - Dictionary

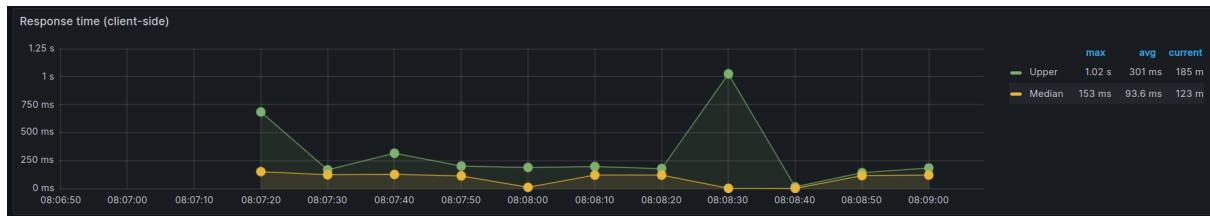


Figure 98: Rate Limiting - Carga - Dictionary

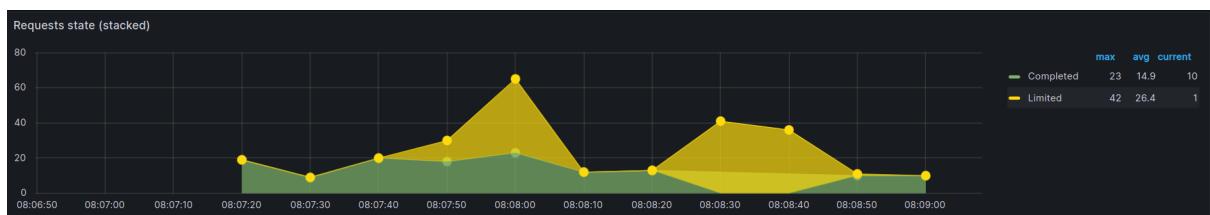


Figure 99: Rate Limiting - Carga - Dictionary



Figure 100: Rate Limiting - Carga - Dictionary

### 6.3.4 Escenario: Carga , Endpoint: /spaceflight\_news

Desde el punto de vista del cliente, no hay una diferencia notable, ya que, en su mayoría, no recibe la respuesta esperada. Sin embargo, desde la perspectiva del proveedor, la implementación de esta limitación evita que el sistema colapse y deje de responder de manera permanente.

Con esta estrategia, se logra una reducción significativa en la utilización de los recursos disponibles.

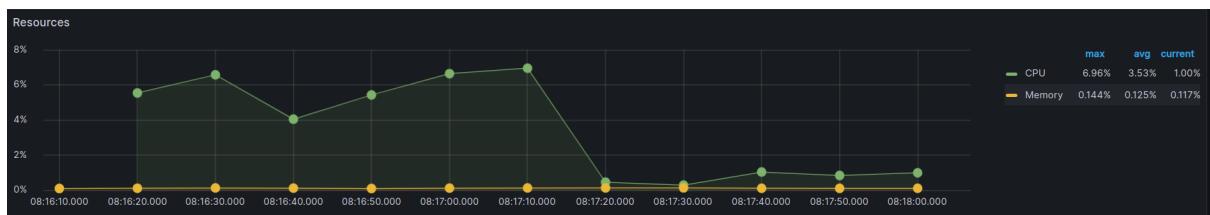


Figure 101: Rate Limiting - Carga - Spaceflight News



Figure 102: Rate Limiting - Carga - Spaceflight News



Figure 103: Rate Limiting - Carga - Spaceflight News

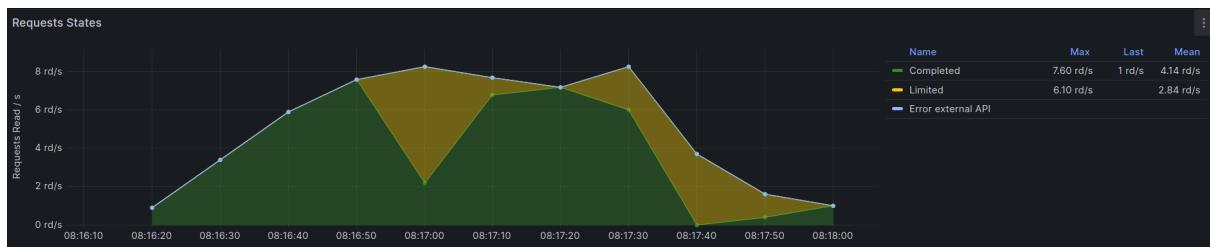


Figure 104: Rate Limiting - Carga - Spaceflight News



Figure 105: Rate Limiting - Carga - Spaceflight News



Figure 106: Rate Limiting - Carga - Spaceflight News

### 6.3.5 Escenario: Estres , Endpoint: /ping

Podemos observar que se alcanzó el límite de solicitudes casi de inmediato al enviar un volumen muy elevado. Después de transcurrido el tiempo de bloqueo, se permitió nuevamente la entrada de solicitudes, lo que generó otro pico menos pronunciado, dado que se encontraba en la fase final del escenario de prueba (en la que se envían 1 solicitud por segundo).

Se evidencia un menor consumo de recursos por parte del contenedor, ya que pasa menos tiempo procesando solicitudes. Esto también permite completar con éxito un mayor número de solicitudes.



Figure 107: Rate Limiting - Estres - Ping

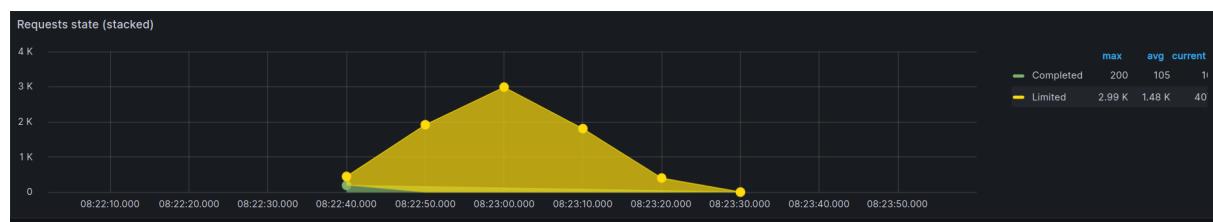


Figure 108: Rate Limiting - Estres - Ping

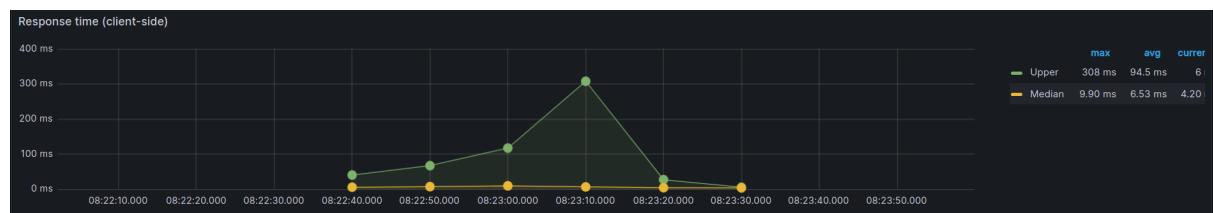


Figure 109: Rate Limiting - Estres - Ping



Figure 110: Rate Limiting - Estres - Ping

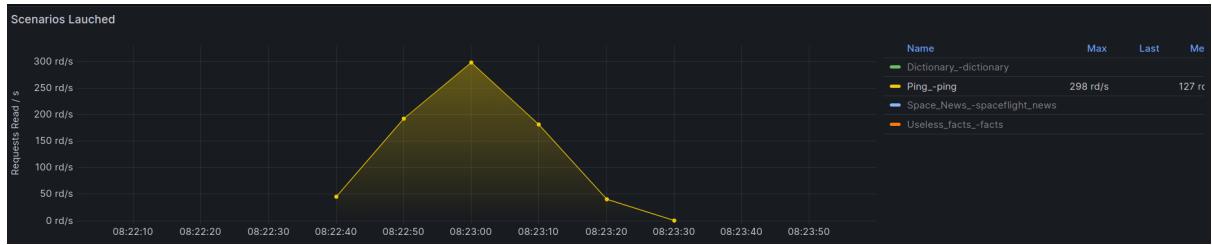


Figure 111: Rate Limiting - Estres - Ping

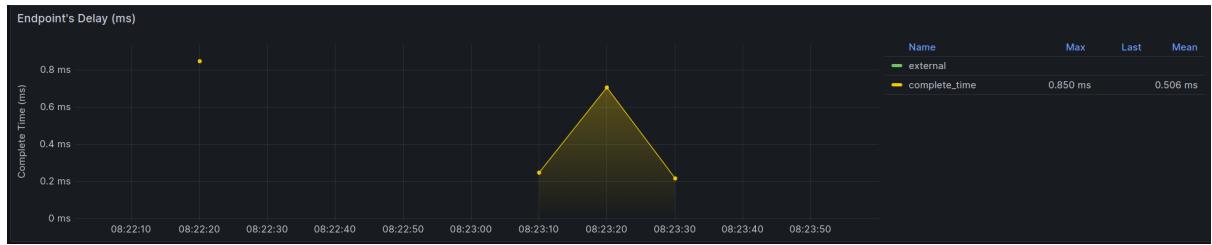


Figure 112: Rate Limiting - Estres - Ping

### 6.3.6 Escenario: Estres , Endpoint: /facts

Al igual que en el caso del ping, se observa que se alcanza el límite de solicitudes casi de inmediato, y luego se recupera una vez transcurrido el tiempo de bloqueo, durante la fase de Stop.

Asimismo, se aprecia una disminución significativa en el consumo de recursos y una salida consistente de respuestas exitosas.

Además, a diferencia del caso base, la implementación de la limitación de solicitudes genera una mejora en la disponibilidad del servicio. Aunque esta limitación interrumpe temporalmente el acceso para la dirección IP que está generando las solicitudes, este periodo es acotado, lo que representa una mejora en comparación con una caída total de la aplicación, ya que el servicio sigue estando disponible para otros clientes.

Con la configuración actual de rate limiting, se observa también una reducción en la cantidad de respuestas por segundo en comparación con el caso base, disminuyendo de un promedio de 20 solicitudes por segundo a aproximadamente 7 solicitudes por segundo.



Figure 113: Rate Limiting - Estres - Facts

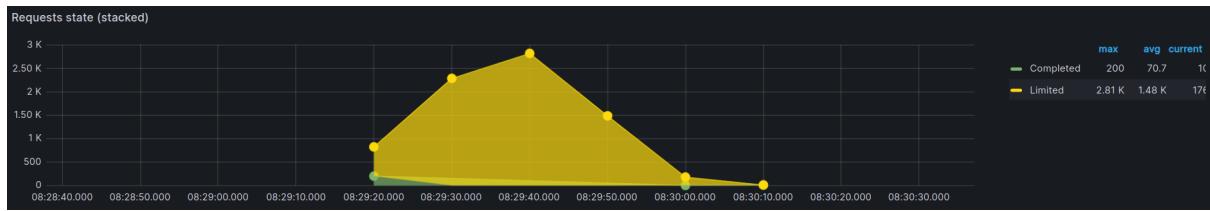


Figure 114: Rate Limiting - Estres - Facts

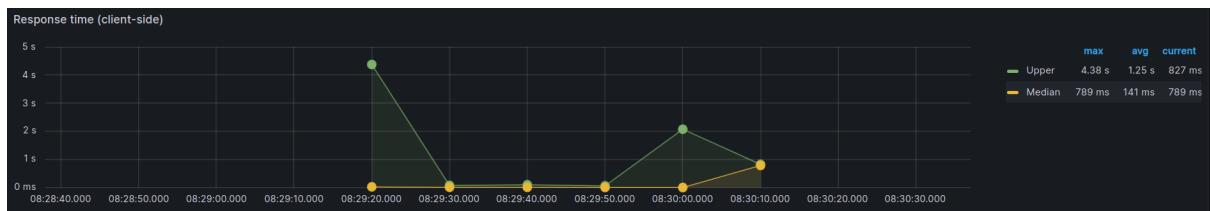


Figure 115: Rate Limiting - Estres - Facts



Figure 116: Rate Limiting - Estres - Facts

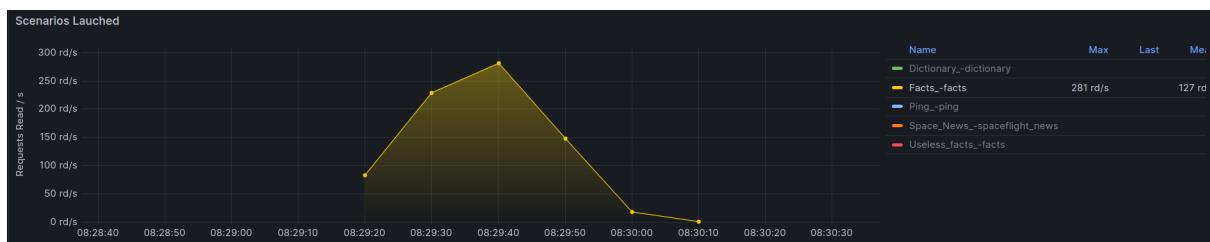


Figure 117: Rate Limiting - Estres - Facts

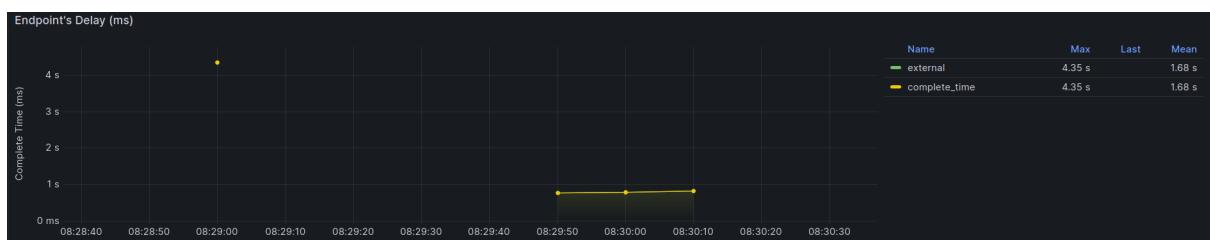


Figure 118: Rate Limiting - Estres - Facts

### 6.3.7 Escenario: Estres , Endpoint: /dictionary

Se observa que se repite el mismo comportamiento que en los otros escenarios para esta táctica. La introducción de la táctica abarca los resultados obtenidos en este escenario.



Figure 119: Rate Limiting - Estres - Dictionary

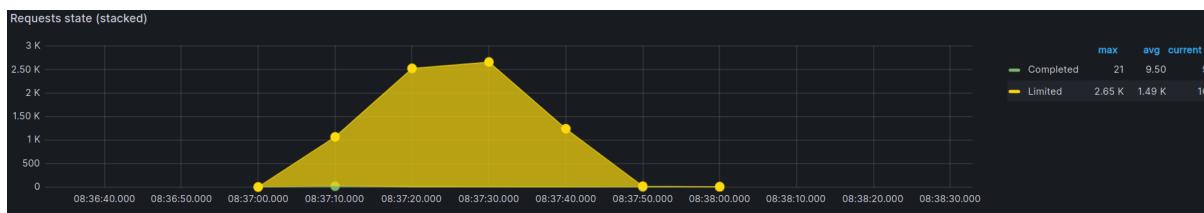


Figure 120: Rate Limiting - Estres - Dictionary



Figure 121: Rate Limiting - Estres - Dictionary

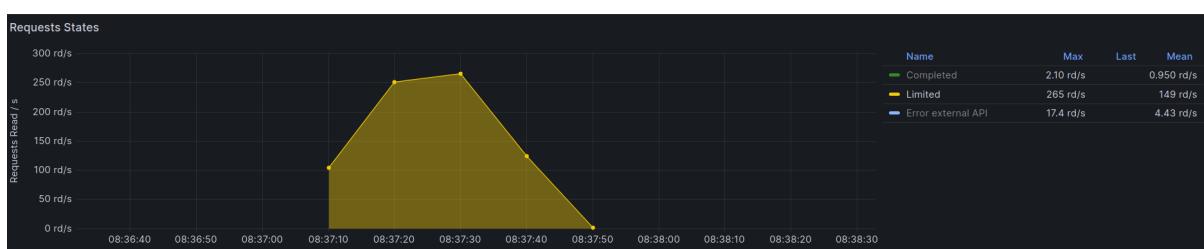


Figure 122: Rate Limiting - Estres - Dictionary



Figure 123: Rate Limiting - Estres - Dictionary

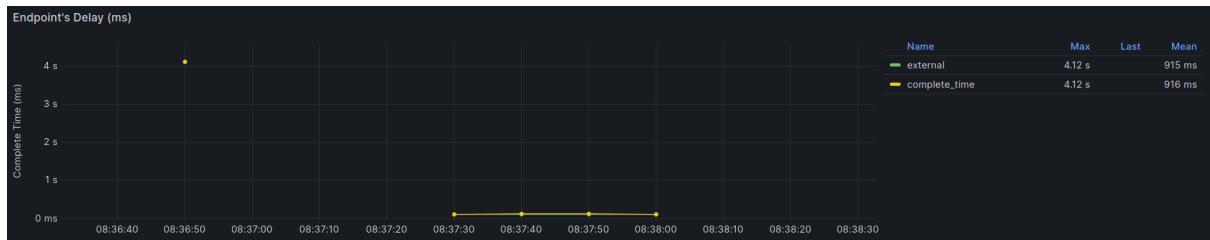


Figure 124: Rate Limiting - Estres - Dictionary

### 6.3.8 Escenario: Estres , Endpoint: /spaceflight\_news

Se observa que se repite el mismo comportamiento que en los otros escenarios para esta táctica. La introducción de la táctica abarca los resultados obtenidos en este escenario.

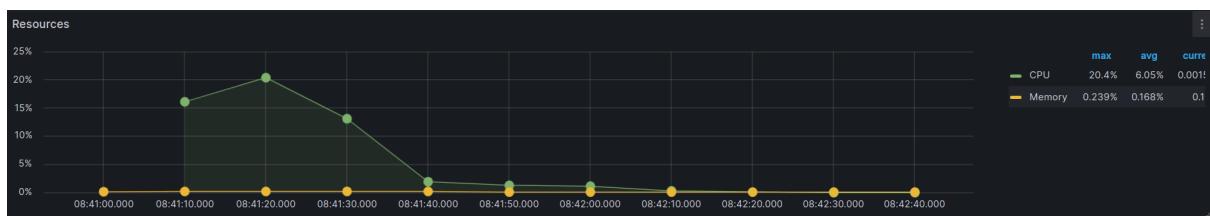


Figure 125: Rate Limiting - Estres - Spaceflight News

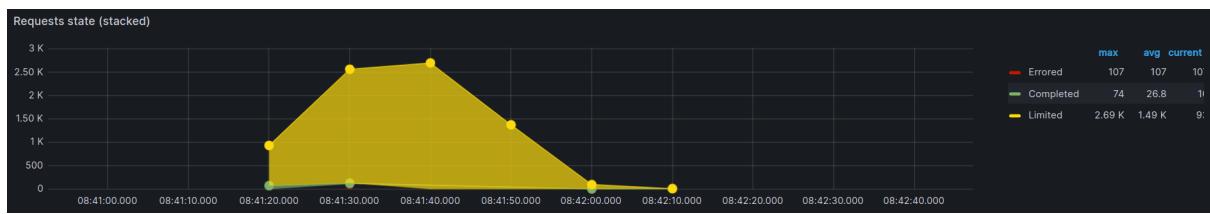


Figure 126: Rate Limiting - Estres - Spaceflight News



Figure 127: Rate Limiting - Estres - Spaceflight News



Figure 128: Rate Limiting - Estres - Spaceflight News



Figure 129: Rate Limiting - Estres - Spaceflight News

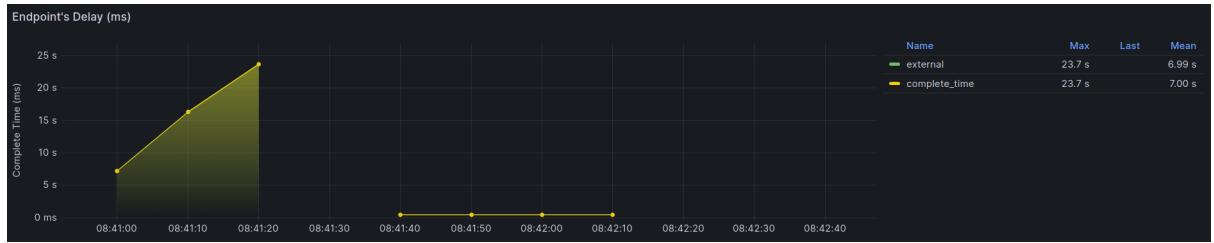


Figure 130: Rate Limiting - Estres - Spaceflight News

## 6.4 Replicación

En esta táctica, se utiliza la replicación del nodo existente en el caso base, generando dos réplicas idénticas. De este modo, Nginx asume el rol de load balancer, distribuyendo las solicitudes entre los tres nodos existentes. A continuación, se muestra como Nginx actua como *load balancer*:

```
[94] pongfederico@Fede:~/Documents/2c24-tp1-arquitectura-de-software/perf$ curl localhost:5555/ping
[23] pongfederico@Fede:~/Documents/2c24-tp1-arquitectura-de-software/perf$ curl localhost:5555/ping
[65] pongfederico@Fede:~/Documents/2c24-tp1-arquitectura-de-software/perf$ curl localhost:5555/ping
[94] pongfederico@Fede:~/Documents/2c24-tp1-arquitectura-de-software/perf$ curl localhost:5555/ping
[23] pongfederico@Fede:~/Documents/2c24-tp1-arquitectura-de-software/perf$ curl localhost:5555/ping
[65] pongfederico@Fede:~/Documents/2c24-tp1-arquitectura-de-software/perf$ curl localhost:5555/ping
```

Figure 131: Replicacion

El numero entre [ ] indica el ID de cada nodo, como se ve hay 3 numeros distintos, por lo que se confirma que Nginx esta distribuyendo las peticiones entre los 3 nodos.

Como conclusión, consideramos que esta táctica no tiene un gran sustento por sí sola, sino que actúa como una táctica complementaria a otras, como es el caso de la caché.

En escenarios de alta demanda, no se observa una diferencia significativa.

### 6.4.1 Escenario: Carga , Endpoint: /ping

Tal como se esperaba, este escenario mostró un rendimiento mucho mejor que en el caso base, ya que las solicitudes se distribuyen entre los distintos nodos, lo que resulta en un menor consumo de recursos. Además, el tiempo de respuesta disminuyó significativamente.



Figure 132: Replicacion - Carga - Ping



Figure 133: Replicacion - Carga - Ping



Figure 134: Replicacion - Carga - Ping



Figure 135: Replicacion - Carga - Ping



Figure 136: Replicacion - Carga - Ping

## Nodo 1



Figure 137: Replicacion - Carga - Ping

## Nodo 2



Figure 138: Replicacion - Carga - Ping

## Nodo 3



Figure 139: Replicacion - Carga - Ping

### 6.4.2 Escenario: Carga , Endpoint: /facts

Tal como se esperaba, este escenario mostró un rendimiento similar al del caso base, pero con una reducción en la cantidad de recursos consumidos. Sin embargo, se observa que el tiempo de respuesta aumentó significativamente. ¿Porque pudo haber pasado esto ultimo?

- Algoritmo de Balanceo Inadecuado: Si el algoritmo de balanceo de carga (como Round Robin) no es adecuado para la carga de trabajo, podría estar enviando solicitudes a nodos que ya están sobrecargados, en lugar de distribuir las solicitudes de manera más equitativa (como con Least Connections). Esto podría aumentar los tiempos de respuesta.
- Falta de sincronización: Si las réplicas no están correctamente sincronizadas, puede haber inconsistencias en el manejo de las solicitudes.
- Entre otros motivos



Figure 140: Replicacion - Carga - Facts



Figure 141: Replicacion - Carga - Facts



Figure 142: Replicacion - Carga - Facts

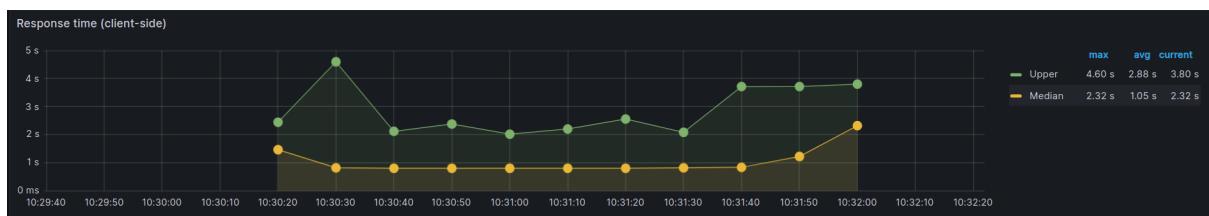


Figure 143: Replicacion - Carga - Facts

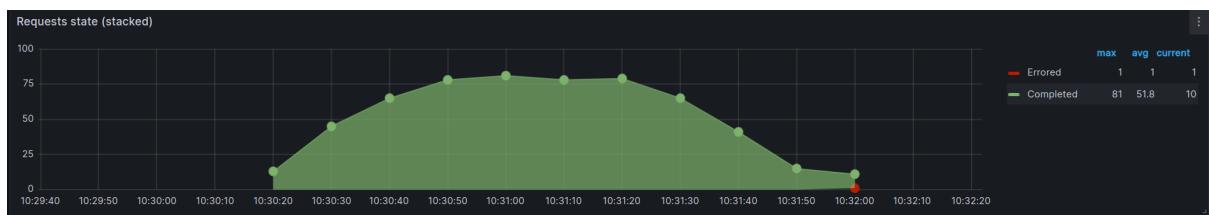


Figure 144: Replicacion - Carga - Facts

## Nodo 1



Figure 145: Replicacion - Carga - Facts

## Nodo 2

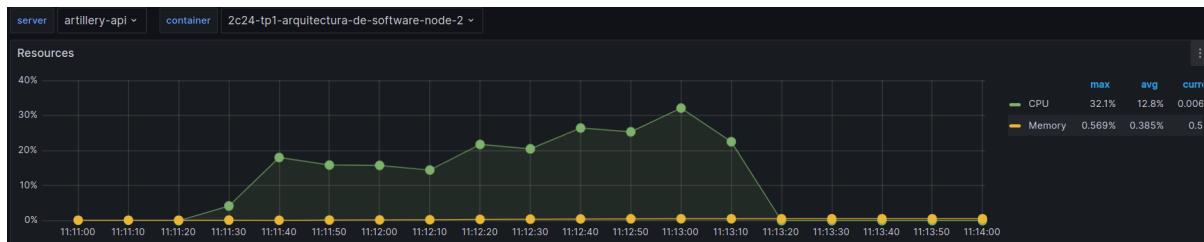


Figure 146: Replicacion - Carga - Facts

## Nodo 3

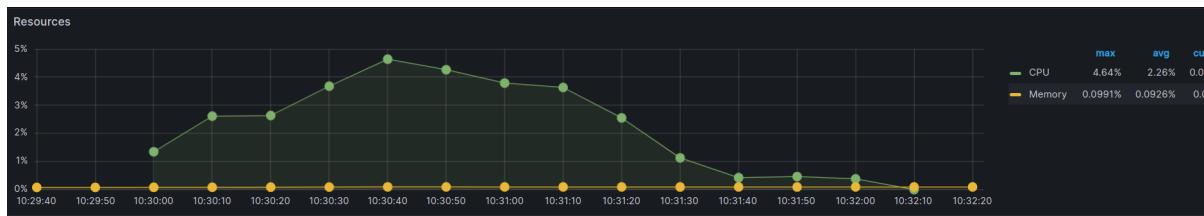


Figure 147: Replicacion - Carga - Facts

### 6.4.3 Escenario: Carga , Endpoint: /dictionary

Se redujo el consumo de recursos del sistema. Sin embargo, no se observan mejoras significativas, ya que, aunque hay múltiples nodos, las consultas a la API externa son realizadas por el mismo cliente, es decir, la misma IP. Por esta razón, las solicitudes siguen estando limitadas.

Esto también se refleja en el tiempo de respuesta, donde se evidencia que no hay una reducción significativa en comparación con el caso anterior.



Figure 148: Replicacion - Carga - Dictionary



Figure 149: Replicacion - Carga - Dictionary

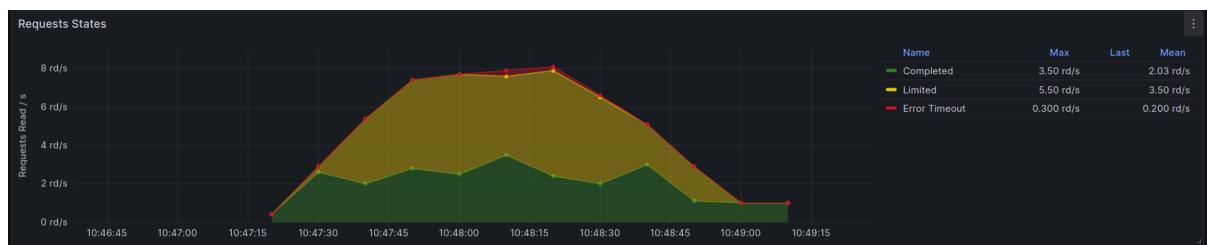


Figure 150: Replicacion - Carga - Dictionary

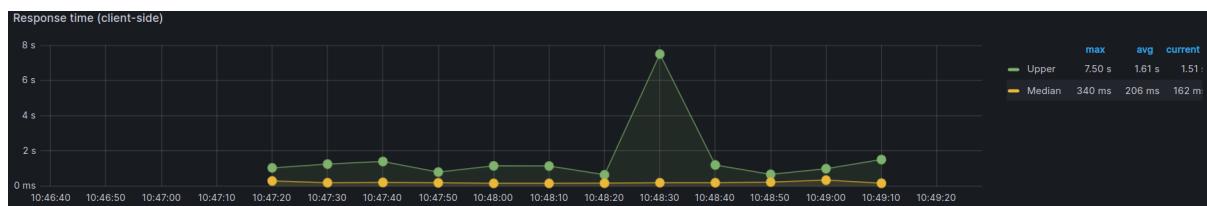


Figure 151: Replicacion - Carga - Dictionary

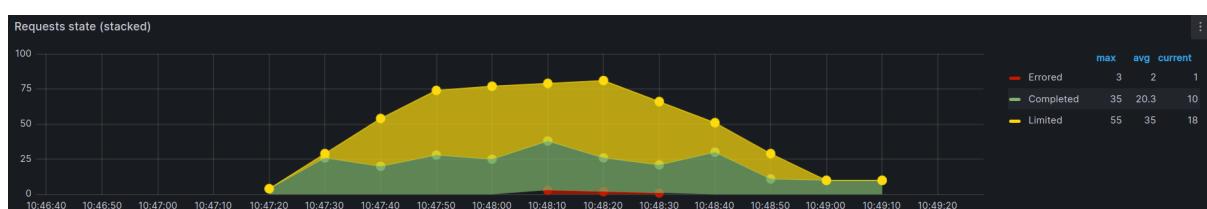


Figure 152: Replicacion - Carga - Dictionary

## Nodo 1

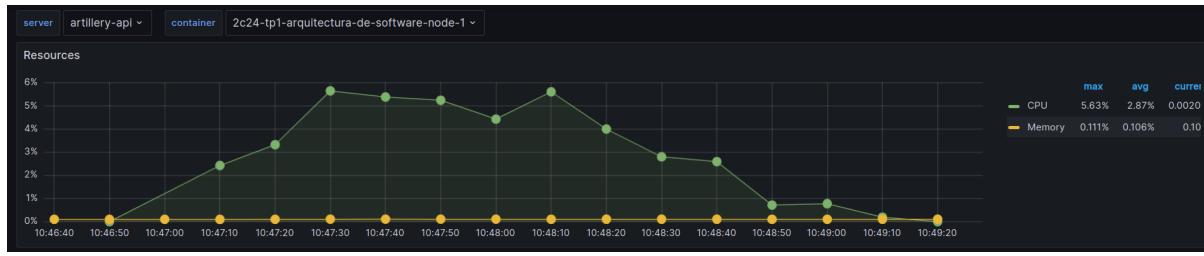


Figure 153: Replicacion - Carga - Dictionary

## Nodo 2



Figure 154: Replicacion - Carga - Dictionary

## Nodo 3

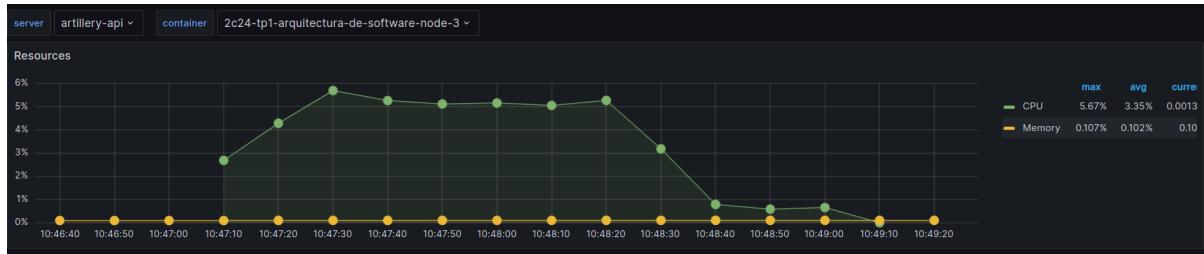


Figure 155: Replicacion - Carga - Dictionary

### 6.4.4 Escenario: Carga , Endpoint: /spaceflight\_news

Tal como se esperaba, dado que en el caso base la prueba resultó exitosa, al replicar el nodo también se logró el mismo resultado. De todas formas, se observó una mejora en términos de utilización de recursos.



Figure 156: Replicacion - Carga - Spaceflight News

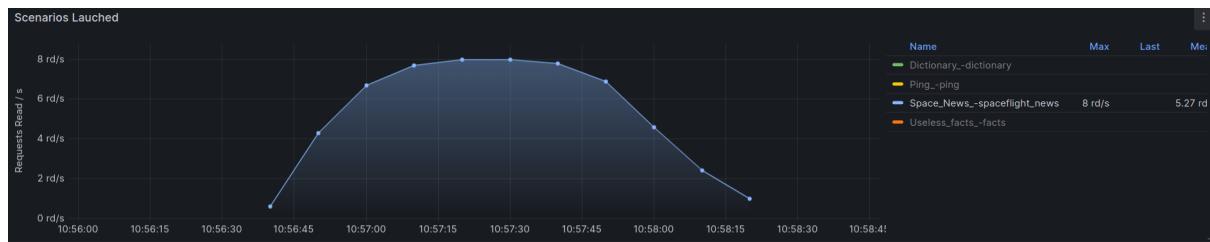


Figure 157: Replicacion - Carga - Spaceflight News

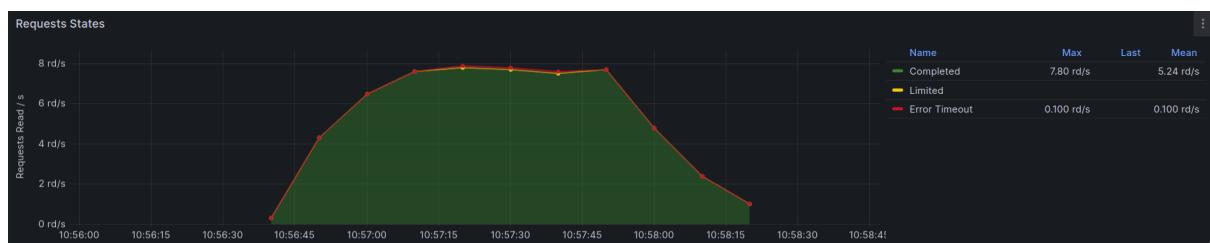


Figure 158: Replicacion - Carga - Spaceflight News

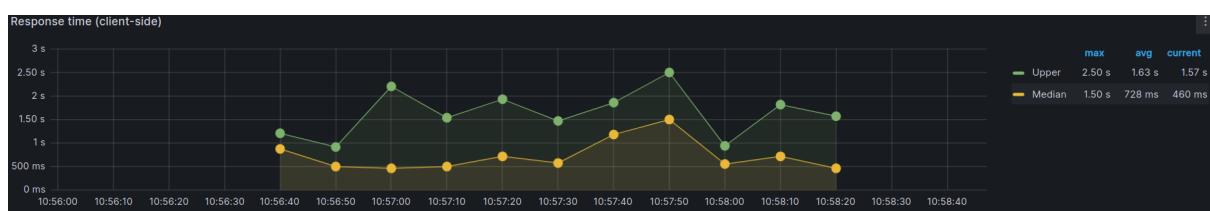


Figure 159: Replicacion - Carga - Spaceflight News

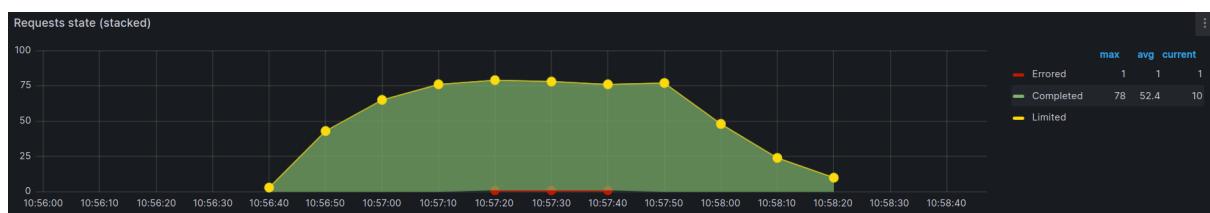


Figure 160: Replicacion - Carga - Spaceflight News

## Nodo 1

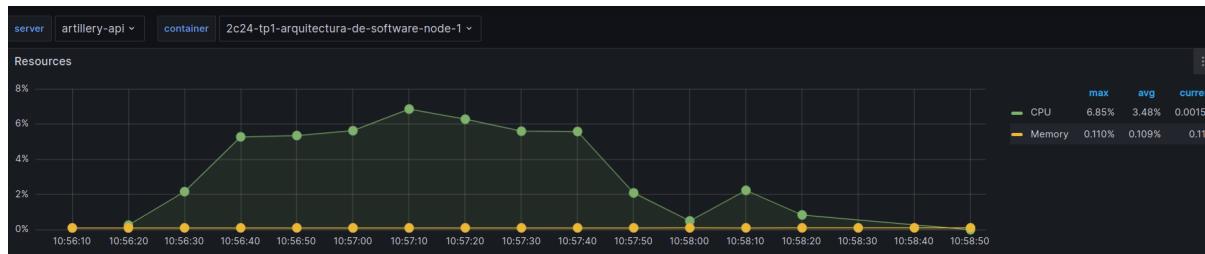


Figure 161: Replicacion - Carga - Spaceflight News

## Nodo 2



Figure 162: Replicacion - Carga - Spaceflight News

## Nodo 3

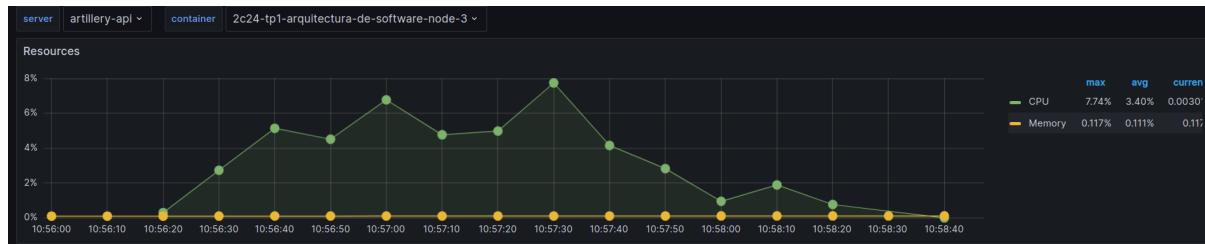


Figure 163: Replicacion - Carga - Spaceflight News

### 6.4.5 Escenario: Estres , Endpoint: /ping

En este escenario, se puede observar que el proceso logra mejorar las fallas experimentadas en el caso base, ya que al reducir el flujo recibido por un nodo, este puede gestionar todas las solicitudes sin inconvenientes. Esto era de esperar, dado que en el caso base se había registrado un porcentaje de fallas muy bajo durante la fase de mayor flujo.

Además, se evidencia una mejora en la utilización de los recursos.

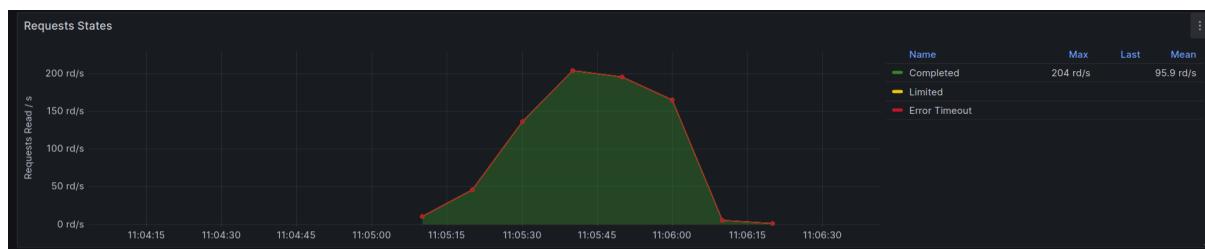


Figure 164: Replicacion - Estres - Ping

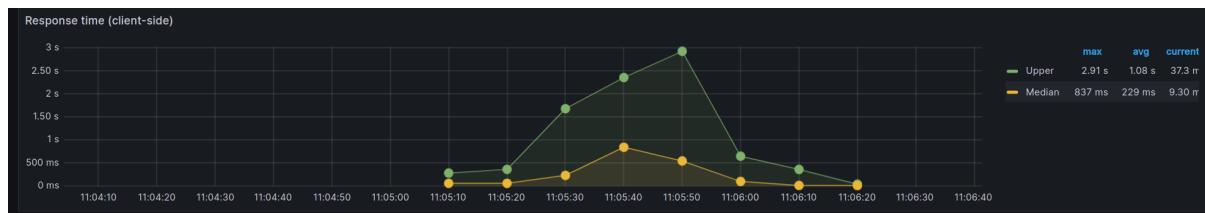


Figure 165: Replicacion - Estres - Ping

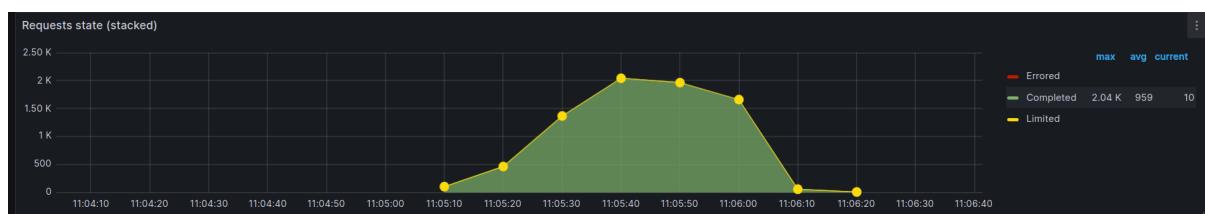


Figure 166: Replicacion - Estres - Ping

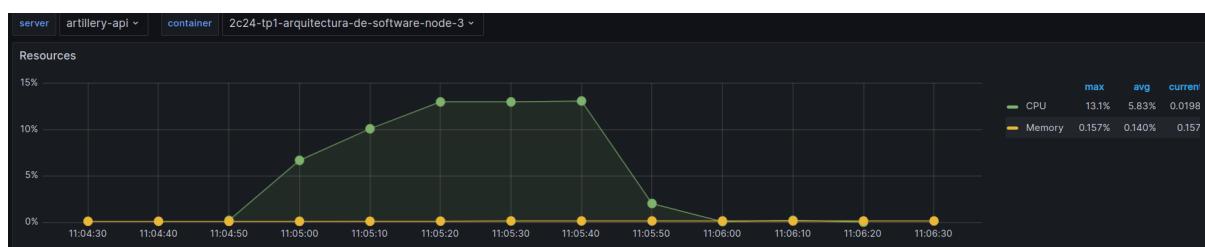


Figure 167: Replicacion - Estres - Ping

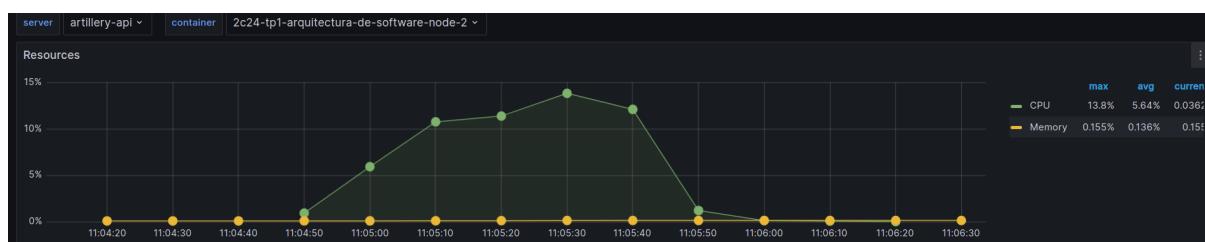


Figure 168: Replicacion - Estres - Ping

## Nodo 1

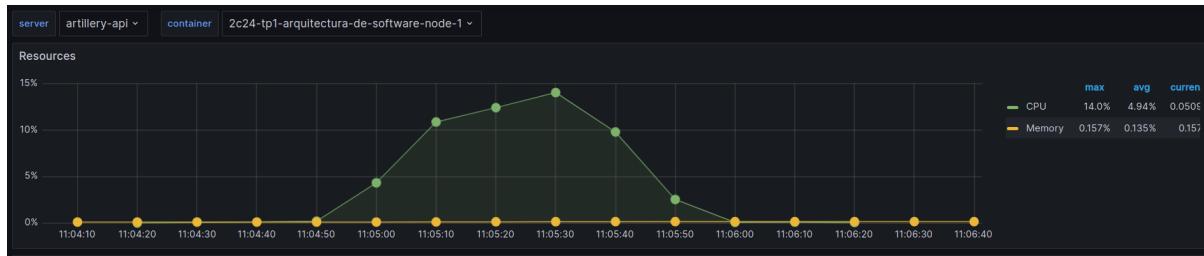


Figure 169: Replicacion - Estres - Ping

## Nodo 2



Figure 170: Replicacion - Estres - Ping

## Nodo 3



Figure 171: Replicacion - Estres - Ping

### 6.4.6 Escenario: Estres , Endpoint: /facts

Se observa una mejora en términos de procesamiento en comparación con el caso base; sin embargo, los resultados obtenidos no son alentadores, dado que la mayoría de las solicitudes son fallidas y solo un pequeño porcentaje resultó exitoso.



Figure 172: Replicacion - Estres - Ping

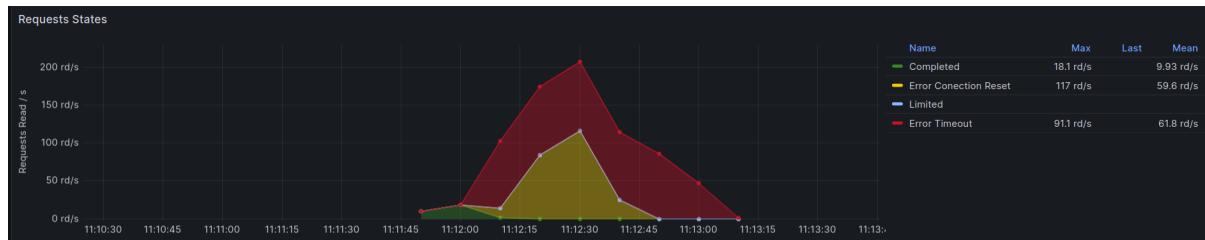


Figure 173: Replicacion - Estres - Ping

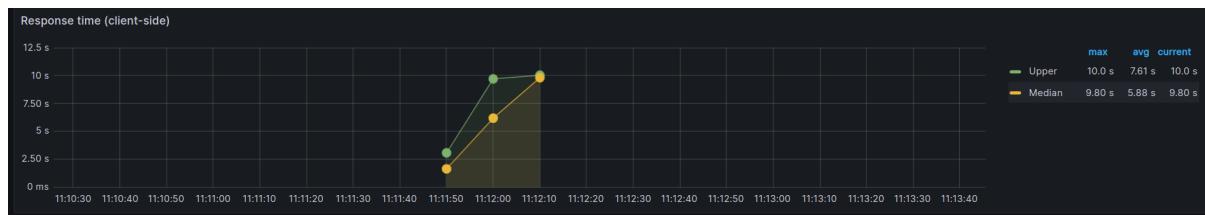


Figure 174: Replicacion - Estres - Ping



Figure 175: Replicacion - Estres - Ping

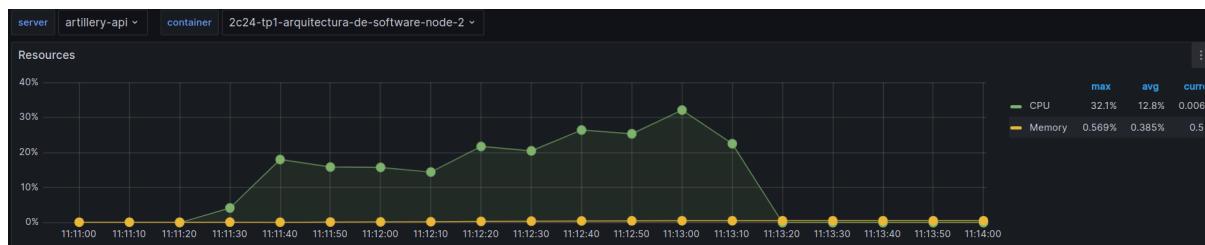


Figure 176: Replicacion - Estres - Ping

## Nodo 1

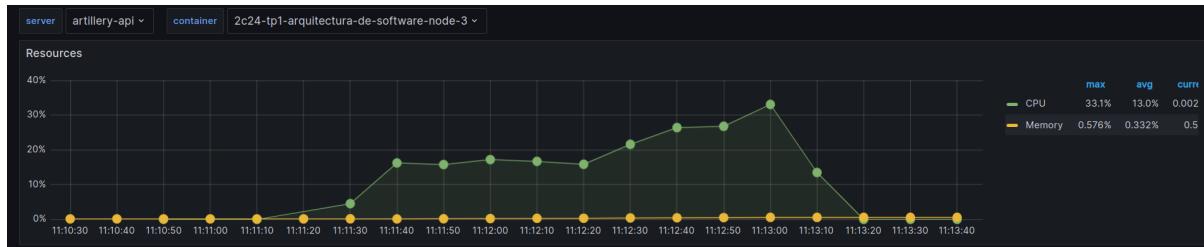


Figure 177: Replicacion - Estres - Ping

## Nodo 2



Figure 178: Replicacion - Estres - Ping

## Nodo 3

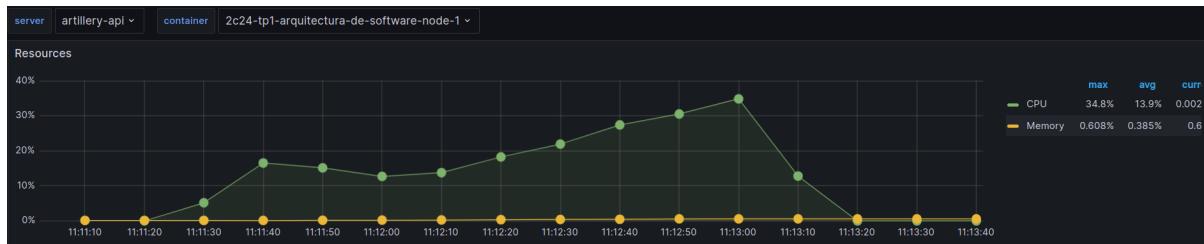


Figure 179: Replicacion - Estres - Ping

### 6.4.7 Escenario: Estres , Endpoint: /dictionary

No se observó una reducción en el consumo de recursos del sistema.

Al igual que en el escenario de carga, no se encuentran mejoras. Esto se debe a que, aunque hay múltiples nodos, la consulta a la API externa la realiza el mismo cliente, es decir, la misma dirección IP. Por esta razón, las solicitudes siguen siendo limitadas.

En ambos escenarios, tanto en el de estrés como en el de carga, la cantidad de solicitudes simuladas es mayor que el límite de solicitudes que recibe la API externa; por lo tanto, ambos casos resultan muy similares, a pesar de que el escenario de estrés tiene un volumen de carga considerablemente superior.

La diferencia entre ambos escenarios radica en que en el caso de estrés se genera una gran cantidad de fallos por timeout debido al elevado volumen de solicitudes.

Esto también se refleja en el tiempo de respuesta obtenido, donde se observa que no hay una reducción significativa en el tiempo.



Figure 180: Replicacion - Estres - Dictionary



Figure 181: Replicacion - Estres - Dictionary

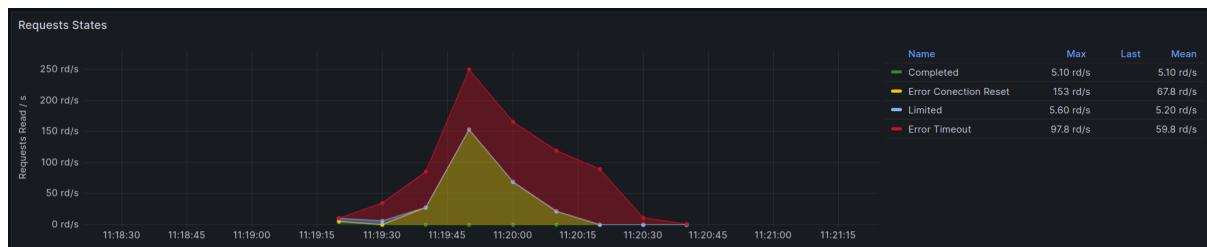


Figure 182: Replicacion - Estres - Dictionary

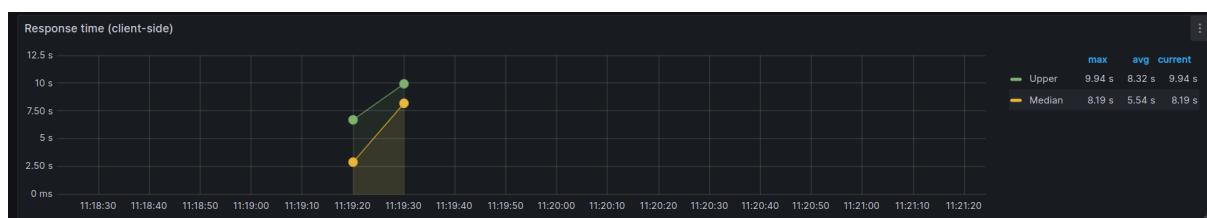


Figure 183: Replicacion - Estres - Dictionary

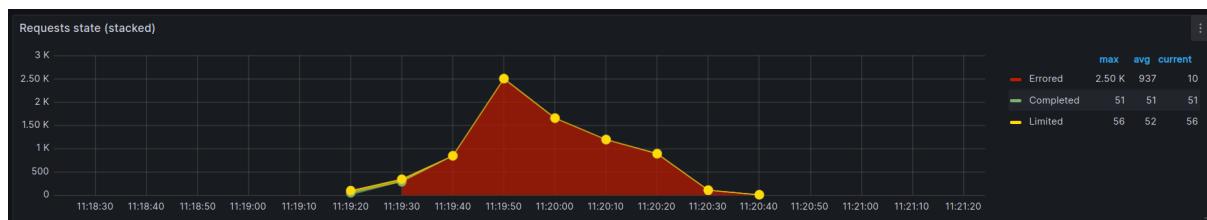


Figure 184: Replicacion - Estres - Dictionary

## Nodo 1

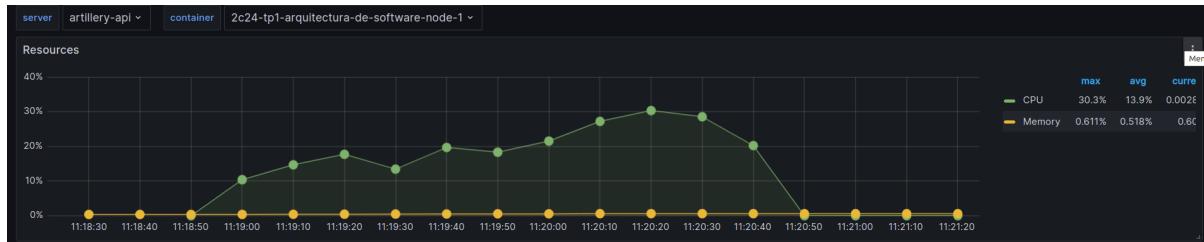


Figure 185: Replicacion - Estres - Dictionary

## Nodo 2

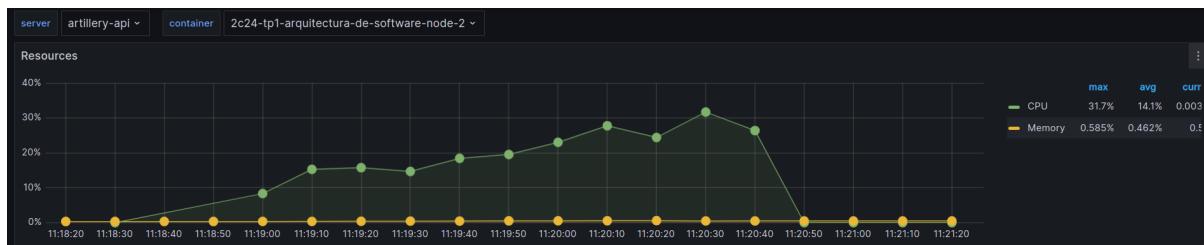


Figure 186: Replicacion - Estres - Dictionary

## Nodo 3

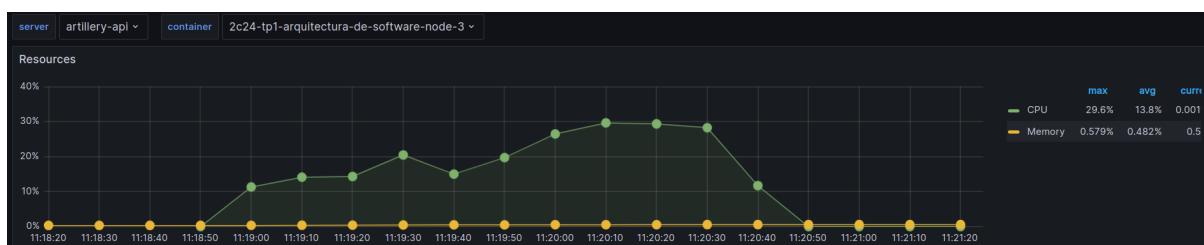


Figure 187: Replicacion - Estres - Dictionary

### 6.4.8 Escenario: Estres , Endpoint: /spaceflight\_news

Con respecto al caso base, la única mejora se observa en términos de consumo de recursos.

Se evidencia que la táctica de replicación de nodos no mejora el rendimiento de la API, ya que al aumentar la cantidad de solicitudes, esta colapsa debido a timeout, resultando en un número muy bajo de solicitudes respondidas con éxito.

El tiempo de respuesta parece estancarse durante la etapa de ramp, a medida que incrementa la cantidad de solicitudes, lo que indica que el sistema no está procesando las solicitudes recibidas.



Figure 188: Replicacion - Estres - Spaceflight News



Figure 189: Replicacion - Estres - Spaceflight News



Figure 190: Replicacion - Estres - Spaceflight News

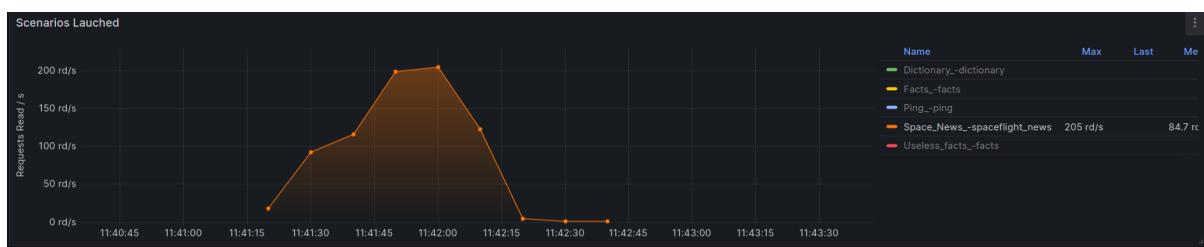


Figure 191: Replicacion - Estres - Spaceflight News

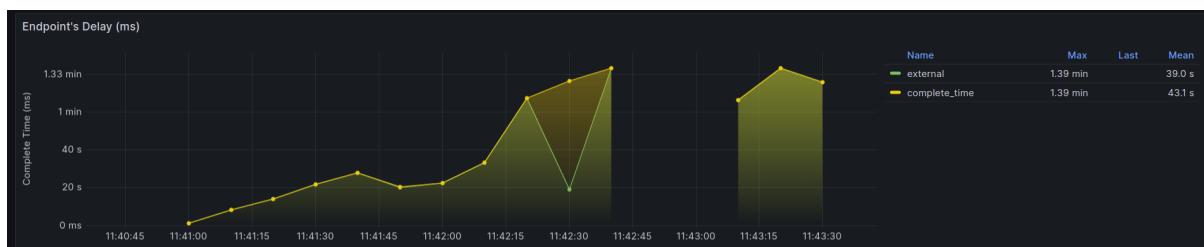


Figure 192: Replicacion - Estres - Spaceflight News

## Nodo 1

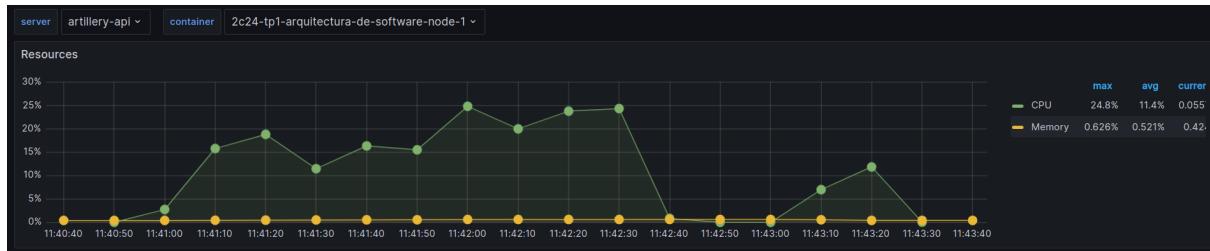


Figure 193: Replicacion - Estres - Spaceflight News

## Nodo 2

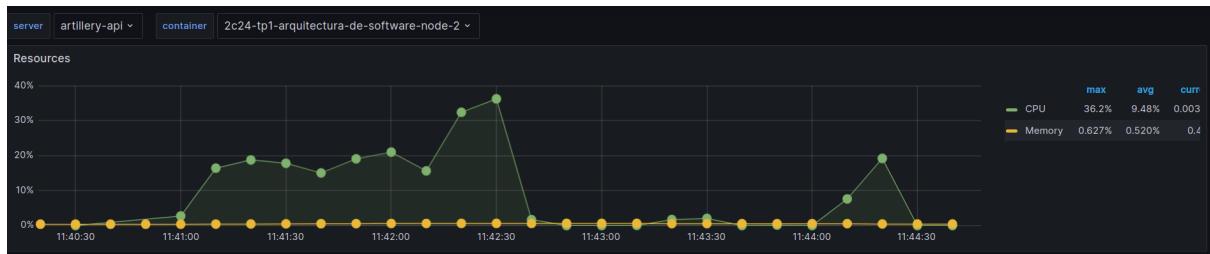


Figure 194: Replicacion - Estres - Spaceflight News

## Nodo 3

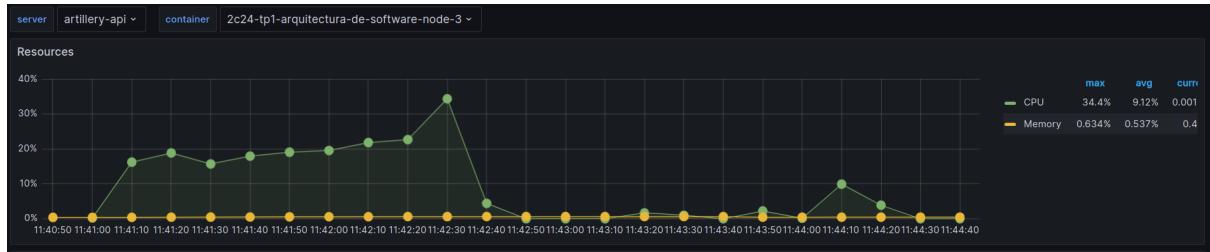


Figure 195: Replicacion - Estres - Spaceflight News

## 7 Conclusiones

Tras implementar cada arquitectura por separado, y haber ejecutado nuestras pruebas en los diferentes endpoints, pudimos analizar y asegurar que cada componente cobra mas valor en un lugar "estratégico". Agregar un componente a nuestra arquitectura cuando no corresponde puede traernos graves problemas en nuestro sistema. Por ejemplo:

### 7.1 /ping

Si bien el endpoint **/ping** se comportó de manera estable en las arquitecturas desarrolladas en el informe, consideramos que lo más conveniente sería aplicar la táctica de rate limiting junto con la de replicación. De esta forma, obtendríamos una mejora en el rendimiento del sistema, asegurando una mayor disponibilidad y un mayor alcance a los clientes en escenarios de alta demanda.

## 7.2 /facts

Para el endpoint **/facts**, creemos que es conveniente aplicar las técnicas de rate limiting y replicación para lograr una mejora en el rendimiento del sistema, aumentando así la disponibilidad y el alcance a clientes en situaciones de alta demanda. ¿Porque no utilizamos por ejemplo la tactica de cache aqui? La tactica de cache no tendría sentido en este endpoint debido a que este endpoint nos devuelve informacion completamente aleatoria en cada request. Por lo que no tiene logica utilizar una cache que almacene informacion aleatoria en cada request, dado que las probabilidades de que en la siguiente request se repita la misma informacion es totalmente baja.

## 7.3 /dictionary

En el caso del endpoint **/dictionary**, sería conveniente implementar las tres tácticas de manera conjunta. Por un lado, tener una caché, que, aunque pueda parecer inusual debido a la gran cantidad de palabras existentes, consideramos utilizar con una capacidad del 1% de las palabras. El diccionario en español contiene aproximadamente 93,000 términos. Tener un porcentaje representativo disminuiría el valor de utilizar una caché, por lo que hemos decidido optar por una muestra pequeña.

Como punto de partida, contemplar el establecimiento de una active population con el 1% de las palabras más utilizadas en español nos parece una idea interesante.

En resumen, entendiendo que existe una vasta cantidad de palabras y que no podemos alojar todas en una caché debido a que carecería de sentido, proponemos esta solución que aprovecha de manera prudente su uso.

Junto con la táctica de caché, podrían aplicarse las tácticas de replicación y rate limiting para mejorar la disponibilidad y el rendimiento.

## 7.4 /spaceflight\_news

Para el endpoint **/spaceflight\_news**, la estrategia óptima sería implementar una táctica de caché, dado que la información requerida se determina por una frecuencia de actualización conocida. Esto permitiría minimizar el número de consultas a la API externa.

No obstante, consideramos que la solución que hemos implementado en la táctica de caché mediante lazy population no es la más adecuada. En su lugar, sería más conveniente utilizar la táctica de active population, la cual actualiza la información de forma periódica, independientemente de si se recibe una solicitud. Esta aproximación resulta eficaz en escenarios donde el volumen de consultas es elevado; sin embargo, en situaciones de bajo tráfico, podría dar lugar a consultas innecesarias.

Adicionalmente, junto con la táctica de caché, podrían implementarse las tácticas de replicación y rate limiting para mejorar tanto la disponibilidad como el rendimiento.