



## Trabajo Práctico N1

Arquitectura de Software

### Integrantes:

**Martin Estrada - 109236**

**Mateo Lardiez - 107992**

**Tomas Vainstein Aranguren - 109043**

**Iara Jolodovsky - 109385**

<b>Trabajo Práctico N1.....</b>	<b>1</b>
Arquitectura de Software.....	1
1. Introducción.....	3
2. Objetivos.....	3
3. Desarrollo.....	4
3.1 Diagrama de componentes - caso base.....	4
3.2 Diagrama de componentes - rate limiting.....	4
3.3 Diagrama de componentes - replicación.....	5
3.4 Diagrama de componentes - Redis.....	5
4. Herramientas de medición.....	7
4.1 Artillery.....	8
¿Qué es y cuál es su objetivo?.....	8
¿Por qué Artillery?.....	8
¿Dónde utilizamos Artillery?.....	8
4.2 StatsD + Graphite + Grafana.....	9
Flujo de monitoreo y visualización.....	9
Métricas visualizadas.....	9
Ejemplo de dashboard.....	9
4.3 cAdvisor.....	9
4.4 Integración general.....	9
5. Ejecución del programa.....	10
6. Nuestras mediciones.....	11
6.1 Caso base.....	11
6.2 Rate Limiting.....	22
6.3 Replicación.....	29
6.4 Redis.....	43
7. Conclusiones.....	52
7.1 Conclusiones generales.....	52

## 1. Introducción

En el presente trabajo práctico se analiza la arquitectura de arVault, un servicio de cambio de divisas desarrollado en Node.js con Express y desplegado en contenedores Docker. La aplicación expone una API REST que permite consultar y actualizar tasas de cambio, administrar cuentas internas y ejecutar operaciones de intercambio de moneda. La infraestructura actual incluye un proxy inverso Nginx y herramientas de observabilidad como StatsD, Graphite, Grafana y cAdvisor, mientras que la persistencia se implementa mediante archivos JSON locales.

Si bien la aplicación cumple con las funcionalidades básicas de un servicio de exchange, presenta diversas limitaciones arquitectónicas: ausencia de transacciones atómicas, almacenamiento no transaccional, punto único de falla, validaciones insuficientes y carencia de mecanismos de seguridad. Estas debilidades impactan directamente en los atributos de calidad (QAs) más críticos para un sistema financiero: confiabilidad, disponibilidad, rendimiento y seguridad.

El objetivo de este informe es proponer, implementar y evaluar tácticas arquitectónicas que permitan mejorar los QAs más relevantes, analizando su efecto sobre la arquitectura base y los trade-offs involucrados.

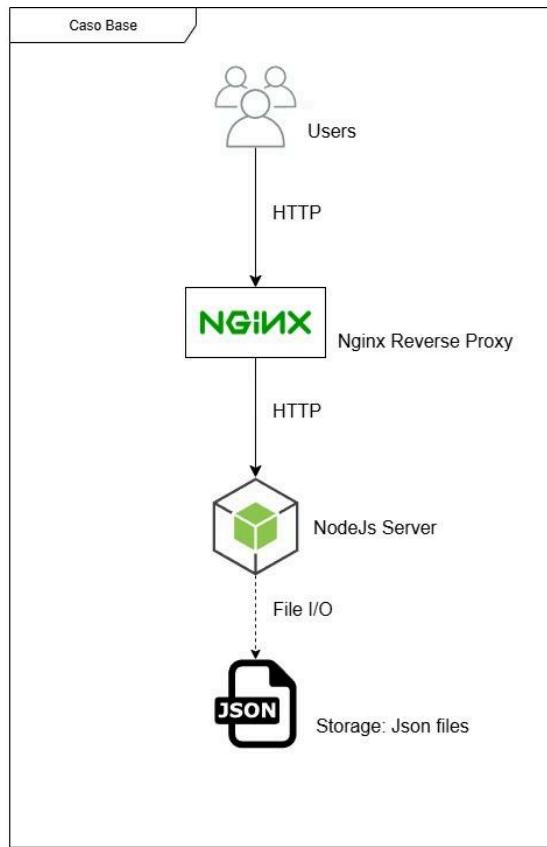
## 2. Objetivos

Los objetivos principales de este trabajo son:

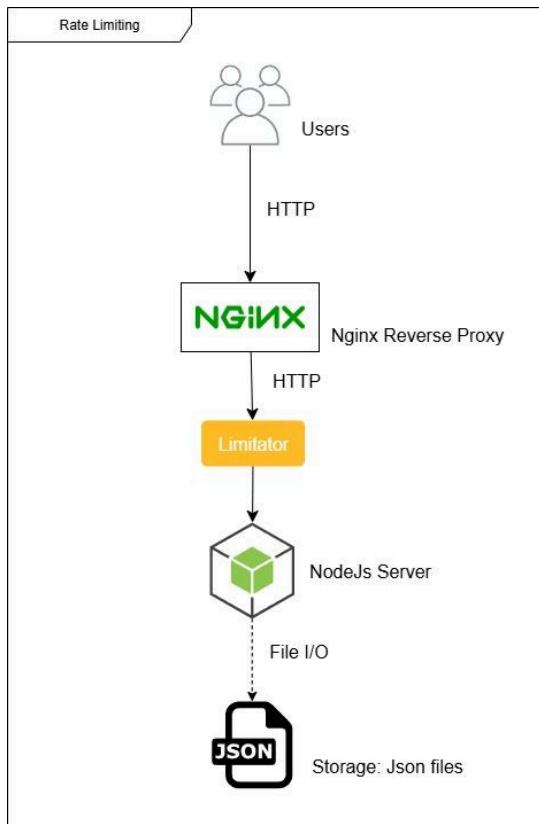
- Analizar la arquitectura base de arVault identificando sus debilidades respecto de los atributos de calidad críticos en un servicio de exchange financiero.
- Seleccionar tácticas arquitectónicas relevantes para mejorar QAs prioritarios como confiabilidad, disponibilidad, rendimiento y seguridad.
- Implementar escenarios de prueba y medición, utilizando herramientas de carga (Artillery) y monitoreo (Graphite + Grafana), con el fin de observar el impacto de cada táctica.
- Comparar el comportamiento entre el caso base y las arquitecturas mejoradas, evaluando beneficios y costos de cada decisión.
- Reflexionar sobre los trade-offs arquitectónicos, reconociendo que no todas las tácticas se orientan a rendimiento, sino también a atributos críticos como seguridad y disponibilidad.

### 3. Desarrollo

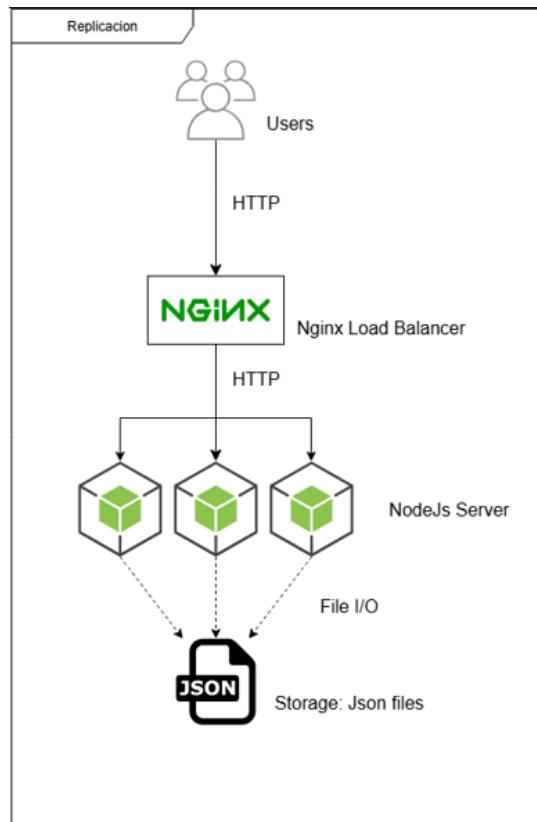
#### 3.1 Diagrama de componentes - caso base



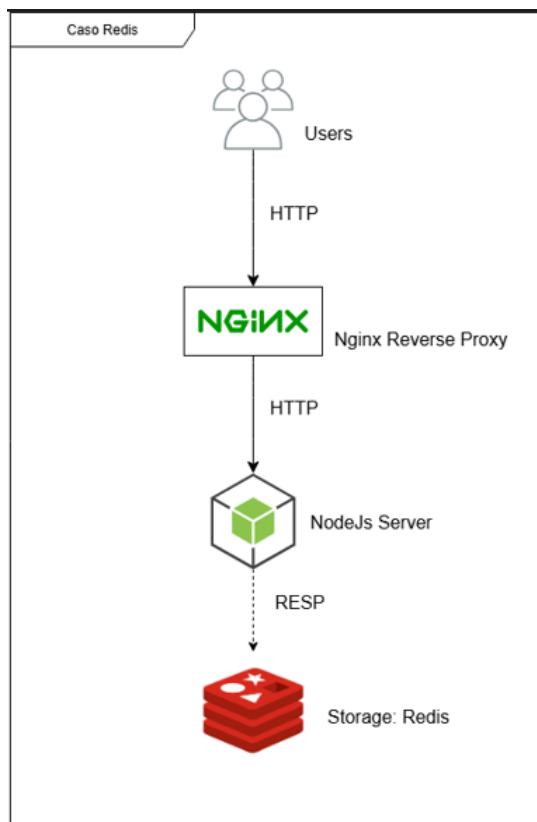
#### 3.2 Diagrama de componentes - rate limiting



### 3.3 Diagrama de componentes - replicación



### 3.4 Diagrama de componentes - Redis



### **3.2 Hipótesis**

En el contexto de arVault, un servicio de exchange financiero simplificado, la elección de tácticas arquitectónicas no es trivial. Cada decisión técnica tiene un impacto directo en los atributos de calidad más críticos para el dominio: confiabilidad, disponibilidad, rendimiento y seguridad.

Muchas veces se asume que siempre existe una “mejor práctica” universal, pero la realidad es que cada táctica implica un trade-off: mejorar un aspecto puede traer costos en otro. Por ejemplo, incorporar caché puede acelerar las respuestas, pero introduce el riesgo de servir datos desactualizados; replicar instancias aumenta disponibilidad, pero complejiza la sincronización de estados; limitar la cantidad de requests protege contra abusos, pero puede afectar la experiencia de usuarios legítimos en escenarios de alta demanda.

En este trabajo buscamos explorar precisamente esas tensiones. La hipótesis central es que, mediante la aplicación de tácticas puntuales como replicación, cache y rate limiting, es posible alcanzar un mejor equilibrio entre los atributos de calidad críticos, sin pretender que exista una única solución óptima.

De este modo, algunas preguntas que guiarán el análisis son:

- ¿Hasta qué punto la incorporación de un caché mejora realmente el rendimiento sin afectar la confiabilidad de los datos?
- ¿La replicación de servicios es suficiente para garantizar disponibilidad, o genera nuevas fuentes de inconsistencia?
- ¿El rate limiting es una protección efectiva contra abusos sin perjudicar a usuarios legítimos?

Estas interrogantes serán discutidas y analizadas en detalle en las secciones siguientes, donde se evaluará el comportamiento del sistema con y sin la aplicación de estas tácticas, utilizando métricas observables como tiempos de respuesta y disponibilidad frente a fallos.

### **Arquitectura - Caso Base**

La arquitectura inicial de arVault se compone de una API monolítica en Node.js + Express, desplegada en un contenedor Docker detrás de un proxy Nginx. Toda la información de cuentas, tasas y logs se persiste en archivos JSON locales (accounts.json, rates.json, log.json).

- Ventajas: simplicidad de despliegue y mantenimiento, facilidad de pruebas.
- Limitaciones: punto único de falla, ausencia de transacciones atómicas, baja confiabilidad en concurrencia, rendimiento limitado por accesos a disco.

### **Arquitectura - Con rate limiting**

En esta variante se añade un middleware de control de tráfico a nivel de API. El sistema define un umbral de requests por cliente en una ventana temporal.

- Cambios principales:
  - Configuración de límites por dirección IP.
  - Respuesta estándar HTTP 429 (“Too Many Requests”) ante exceso.
- Impacto esperado:
  - Seguridad/Disponibilidad: mitigación de ataques de denegación de servicio (DoS) y uso abusivo.

- Usabilidad: usuarios legítimos pueden experimentar bloqueos en picos de carga.

### **Arquitectura - Con replicacion de servicios**

Esta implementación introduce replicación mediante una arquitectura de múltiples instancias, orquestadas con Docker Compose para ejecutar tres réplicas del servicio API en contenedores independientes que comparten el mismo código base.

Cambios principales:

- Configuración de Nginx como reverse proxy para balancear las solicitudes entrantes entre las tres instancias (exchange-api-1, exchange-api-2, exchange-api-3) utilizando el algoritmo *round-robin* por defecto.

Impacto esperado:

- Alta disponibilidad: continuidad del servicio ante fallos de instancias individuales.
- Escalabilidad horizontal: capacidad de manejar hasta tres veces más carga que una implementación monoinstancia.
- Performance: mayor throughput y reducción de latencia gracias a la paralelización del procesamiento.
- Resiliencia: tolerancia a fallos sin interrupción del servicio.

### **Arquitectura - Con Redis**

Esta implementación reemplaza el almacenamiento en archivos JSON por Redis. Las operaciones de lectura de tasas se resuelven directamente desde la Redis, reduciendo la dependencia del almacenamiento en archivos.

- Cambios principales:
  - Integración de un servicio de Redis como una base de datos en memoria persistente que reemplaza el uso de archivos JSON.
- Estrategia de *write-through* o time-to-live (TTL) para mantener la coherencia de los datos.
- Impacto esperado:
  - Performance: Acceso en memoria vs I/O de disco
  - Concurrencia: Redis maneja operaciones atómicas nativamente
  - Escalabilidad: Servicio dedicado de almacenamiento
  - Persistencia: Los datos se mantienen entre reinicios (configuración por defecto)

## **4. Herramientas de medición**

El análisis de los atributos de calidad de un sistema requiere una instrumentación adecuada que permita **observar, registrar y comparar** métricas de comportamiento bajo diferentes condiciones de carga.

Para este trabajo, se utilizaron múltiples herramientas integradas en la arquitectura de despliegue de

*arVault*, con el objetivo de obtener una visión completa del rendimiento, disponibilidad y volumen de operaciones en tiempo real.

Las principales herramientas empleadas fueron:

## 4.1 Artillery

### ¿Qué es y cuál es su objetivo?

**Artillery** es una herramienta de *load testing* y *performance testing* para servicios web. Permite simular múltiples usuarios concurrentes que ejecutan solicitudes HTTP hacia un servidor, generando una carga configurable y registrando métricas detalladas sobre tiempos de respuesta, errores, throughput y consumo de recursos.

Su propósito es **evaluar el comportamiento del sistema bajo diferentes escenarios de carga y estrés**, identificando cuellos de botella, latencias inesperadas y degradaciones de servicio.

### ¿Por qué Artillery?

Se eligió Artillery por su  **simplicidad de configuración**, su compatibilidad con entornos Node.js y su capacidad de integrarse fácilmente con herramientas de monitoreo mediante **StatsD y Graphite**.

Permite definir escenarios de prueba precisos con múltiples fases (*ramp-up*, *steady-state* y *ramp-down*), ajustando la intensidad del tráfico de acuerdo con los casos de estudio.

### ¿Dónde utilizamos Artillery?

Artillery se utilizó para generar tráfico sobre los principales endpoints de la API de arVault:

- GET /accounts
- PUT /accounts/{id}/balance
- GET /rates
- PUT /rates
- GET /log
- POST /exchange

Cada escenario se definió con un **archivo YAML independiente**, utilizando los mismos parámetros de carga y estrés para facilitar la comparación entre tácticas:

- **Carga:**  
Cuatro fases de 30 s cada una: *ramp-up* (2 → 8 req/s), *plain* (8 req/s), *ramp-down* (8 → 2 req/s) y *stop* (1 req/s).  
Este escenario representa un uso normal y sostenido del sistema.
- **Estrés:**  
Cuatro fases de 20–30 s: *ramp-up* (100 → 500 req/s), *plain* (600 req/s), *ramp-down* (500 → 100 req/s) y *stop* (1 req/s).  
Este escenario busca evaluar la tolerancia y degradación ante **sobrecarga extrema**.

Los escenarios fueron configurados para enviar las métricas a través del **plugin de StatsD**, con un prefijo diferente por caso (por ejemplo, *artillery-api-base* o *artillery-api-rl*), permitiendo visualizar los resultados en dashboards separados dentro de Grafana.

## 4.2 StatsD + Graphite + Grafana

### Flujo de monitoreo y visualización

Para observar en tiempo real las métricas de rendimiento, se integró la pila **StatsD → Graphite → Grafana**.

Artillery envía las métricas generadas (tiempos de respuesta, latencias, tasas de error, throughput, etc.) al daemon de **StatsD**, el cual las reenvía a **Graphite** para su almacenamiento temporal en series de tiempo.

Luego, **Grafana** consume esas series y las visualiza en dashboards interactivos, permitiendo correlacionar los resultados de cada escenario con el consumo de recursos del sistema.

### Métricas visualizadas

Los dashboards de Grafana incluyeron:

- **Tiempo de respuesta promedio y percentiles (p50, p95, p99)**
- **Throughput (requests/segundo)**
- **Errores (HTTP 4xx / 5xx / 429)**
- **Uso de CPU y memoria (vía cAdvisor)**
- **Volumen de operaciones por moneda** (sumatoria de compras y ventas)
- **Neto operado** (compras – ventas, acumuladas en el tiempo)

Estas últimas métricas de negocio se incorporaron especialmente para cumplir con el requerimiento del enunciado: mostrar el volumen operado por moneda y su variación a lo largo del tiempo.

### Ejemplo de dashboard

Cada caso (base y con tácticas aplicadas) se asoció a un panel específico en Grafana.

Allí se visualizaron simultáneamente las métricas técnicas (rendimiento y disponibilidad) junto con las métricas de negocio (volumen y neto por moneda).

Esto permitió detectar correlaciones, como picos de CPU coincidentes con momentos de alto volumen transaccional o caídas de throughput durante la activación del *rate limiting*.

## 4.3 cAdvisor

Para observar el consumo de recursos de los contenedores Docker, se utilizó **cAdvisor (Container Advisor)**, una herramienta de monitoreo desarrollada por Google que recolecta métricas de CPU, memoria, disco y red para cada contenedor en ejecución.

Esta información se integró al dashboard de Grafana, lo que permitió **evaluar el impacto de cada táctica arquitectónica sobre el uso de recursos del sistema**.

## 4.4 Integración general

En conjunto, las herramientas mencionadas conforman un entorno de observabilidad completo. Este flujo permitió realizar mediciones cuantitativas y reproducibles sobre el sistema, facilitando la comparación objetiva entre el caso base y los escenarios mejorados (por ejemplo, con rate limiting o caché).

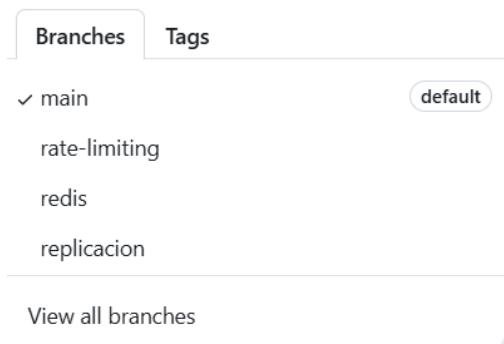
## 5. Ejecución del programa

En esta sección se describen los pasos necesarios para ejecutar el entorno completo de **arVault**, incluyendo la aplicación principal, los servicios auxiliares de monitoreo y las pruebas de carga y estrés.

El objetivo es que cualquier lector pueda **reproducir los resultados y métricas** presentados en este informe.

Antes de ejecutar el sistema, es necesario contar con las siguientes herramientas instaladas en el entorno local:

- **Docker** y **Docker Compose** (versión 20.10 o superior)
- **Node.js** (versión 18 o superior)
- **npm** (para instalación de Artillery y dependencias locales)
- **Git** (para clonar el repositorio del proyecto)



Deberá moverse a la rama elegida.

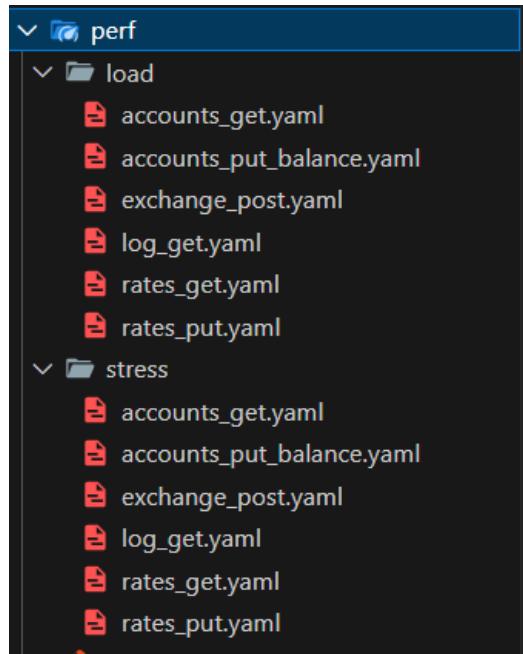
```
git checkout <branch>
```

Y luego levantar los contenedores

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

CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED
88792f36673a	gcr.io/cadvisor/cadvisor:v0.49.2	"/usr/bin/cadvisor -..."	About a minute ago exchange-cadvisor-1	
ede7efa6fe46	nginx:1.27.4	"/docker-entrypoint..."	About a minute ago exchange-nginx-1	
94059cd18439	graphiteapp/graphite-statsd:1.1.10-5	"/entrypoint"	About a minute ago	
126/tcp, :::8126->8126/tcp, 0.0.0.0:8090->80/tcp		[::]:8090->80/tcp	exchange-graphite-1	
d1b922eea92a	exchange-api	"node app.js"	About a minute ago exchange-api-1	
0df742db3efb	grafana/grafana:11.5.2	"/run.sh"	About a minute ago exchange-grafana-1	

En una nueva terminal deberá moverse a la carpeta /perf donde se pueden visualizar los escenarios de load y stress



Vamos a elegir el endpoint que queremos probar y su escenario y con la herramienta artillery se ejecuta:

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

## 6. Nuestras mediciones

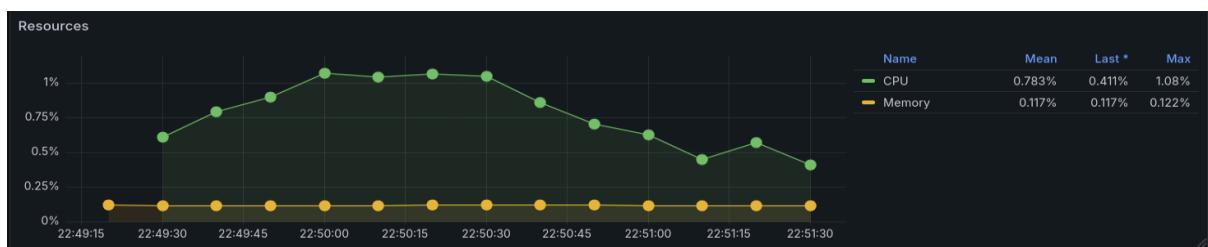
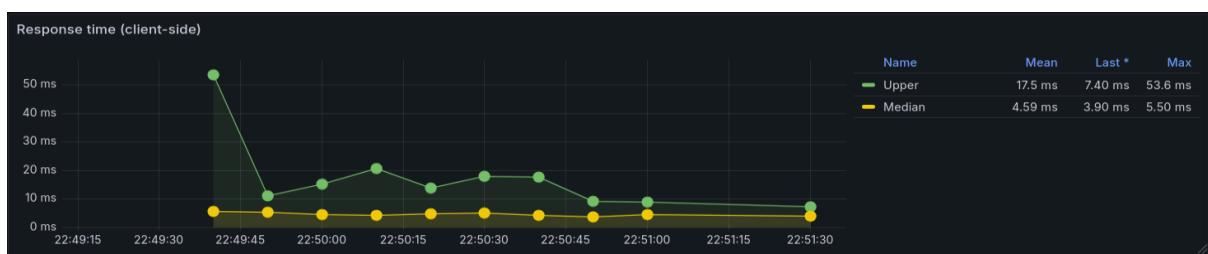
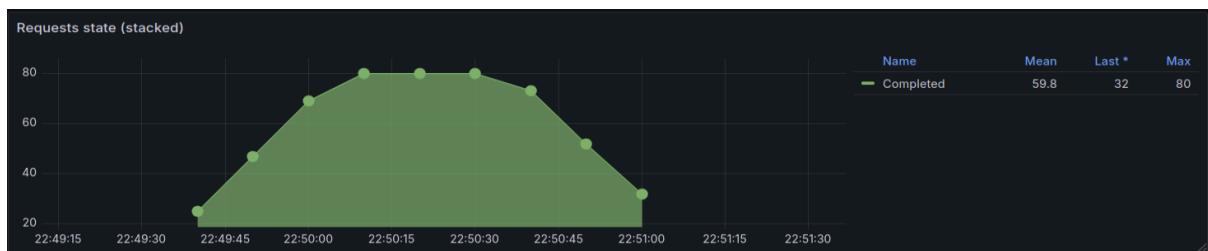
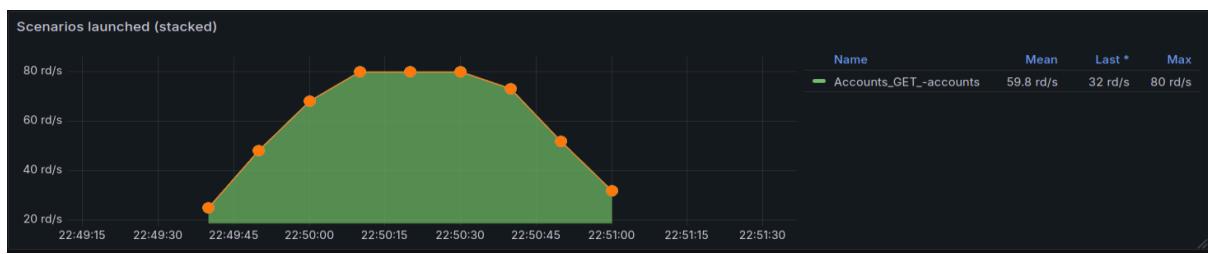
### 6.1 Caso base

**Escenario: Carga, Endpoint: GET /accounts**

Durante la prueba de carga, el tráfico aumentó progresivamente de 2 a 8 req/s, alcanzando un máximo de 80 req/s y un promedio de 59,8 req/s, sin errores ni interrupciones.

El sistema manejó correctamente todas las solicitudes, mostrando latencias muy bajas y estables, con una mediana de 4,6 ms y un pico máximo de 53,6 ms al inicio del *ramp-up*.

El consumo de recursos fue mínimo: CPU promedio 0,78 % (pico 1,08 %) y memoria 0,12 %, evidenciando que la operación de lectura de cuentas es liviana y que el servicio mantiene un rendimiento óptimo bajo carga sostenida.

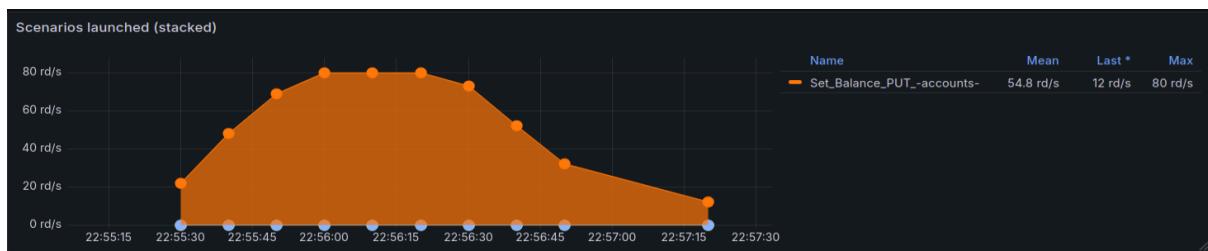


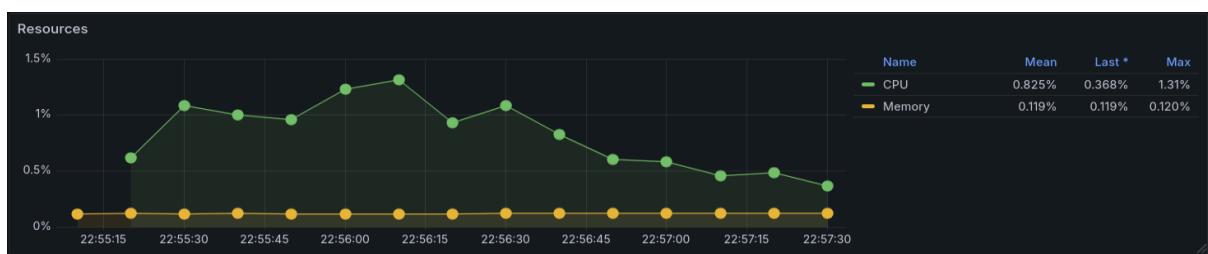
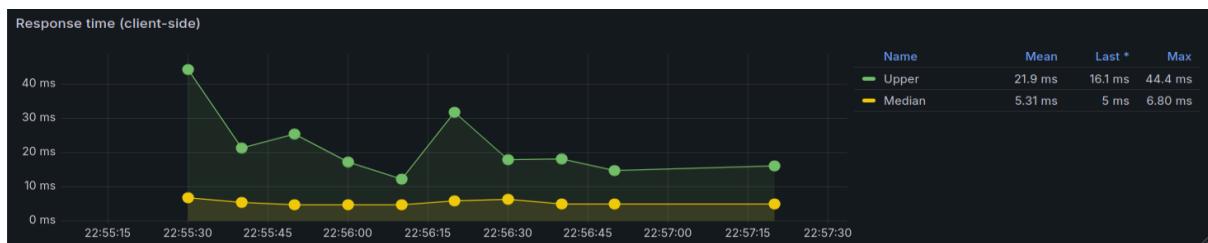
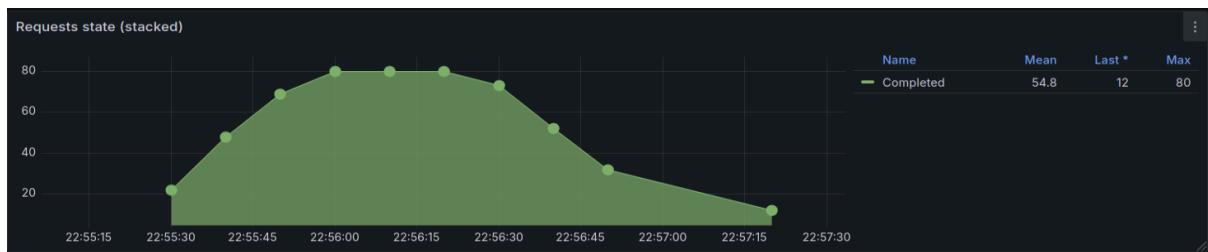
### Escenario: Carga, Endpoint: PUT /accounts/{id}/balance

El endpoint `PUT /accounts/{id}/balance`, al realizar escrituras sobre el almacenamiento local, mostró un máximo de 80 req/s y un promedio de 54,8 req/s, sin errores ni interrupciones.

Las latencias se mantuvieron bajas (mediana 5,3 ms, pico 44,4 ms), con un leve aumento respecto a las operaciones de lectura, propio de las actualizaciones en disco.

El consumo de recursos fue estable (CPU promedio 0,83 %, memoria 0,12 %), lo que demuestra que el endpoint maneja eficientemente las operaciones de escritura sin afectar el rendimiento general del sistema.



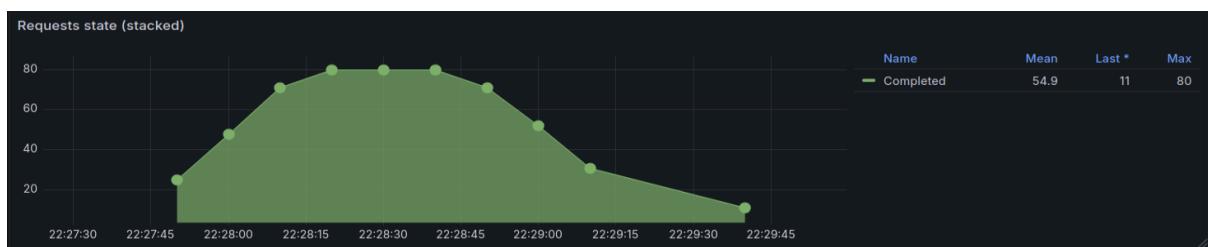
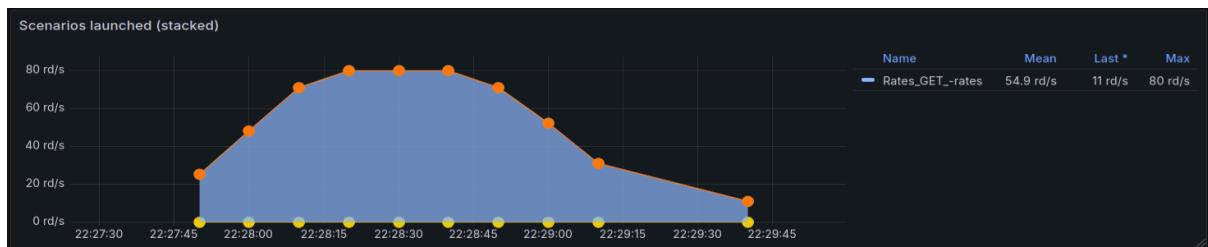


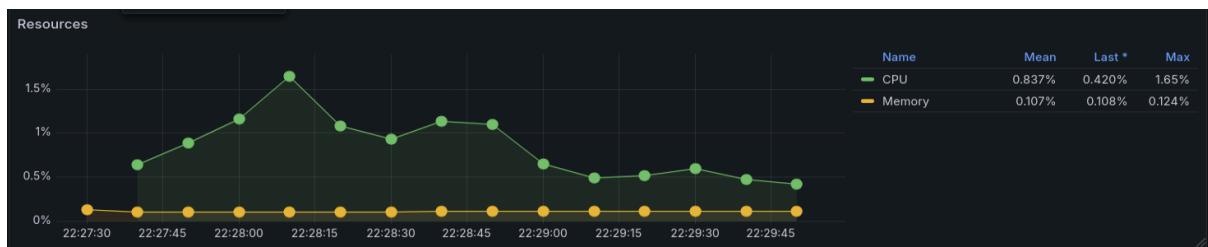
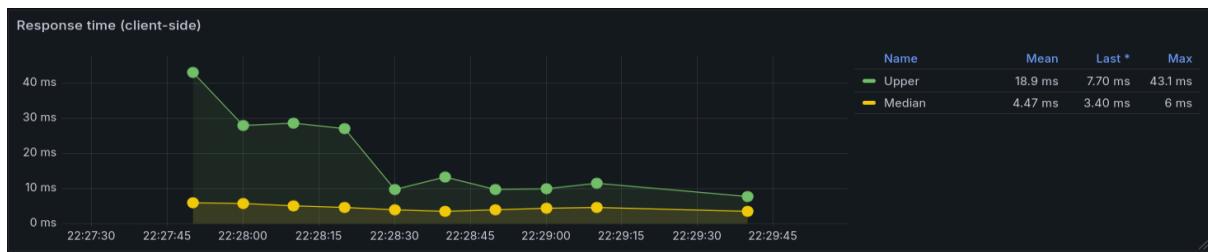
### Escenario: Carga, Endpoint: GET /rates

El endpoint *GET /rates*, que obtiene las tasas de cambio mantenidas en memoria, mostró un máximo de 80 req/s y un promedio de 54,9 req/s, sin errores ni interrupciones.

Las latencias fueron muy bajas (mediana 4,5 ms, pico 43,1 ms), lo que confirma la eficiencia de la lectura en memoria.

El uso de recursos fue mínimo (CPU promedio 0,83 %, memoria 0,10 %), evidenciando que esta operación tiene un impacto prácticamente nulo en el rendimiento general del sistema y una alta estabilidad bajo carga moderada.





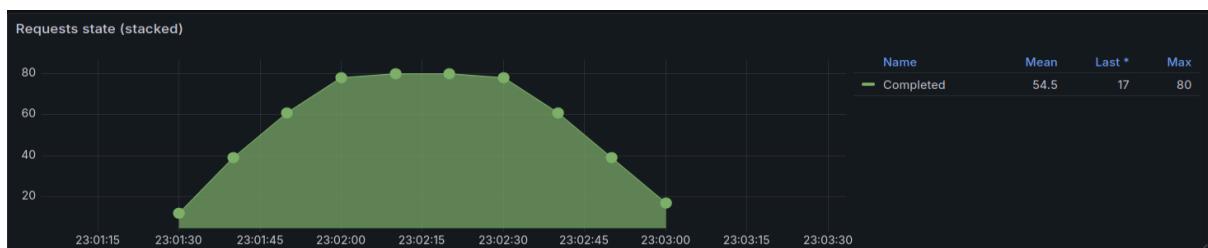
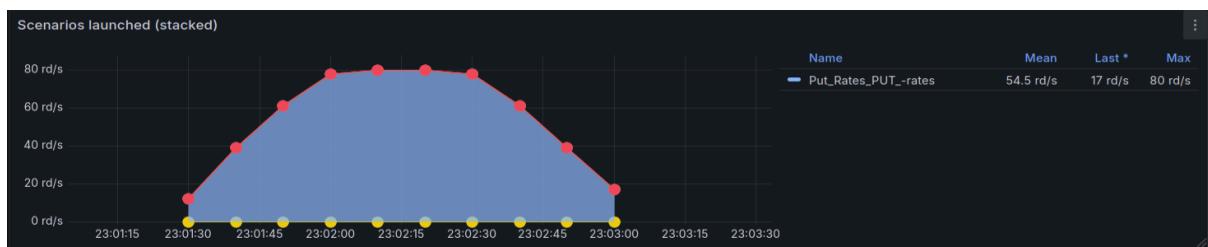
### Escenario: Carga, Endpoint: PUT /rates

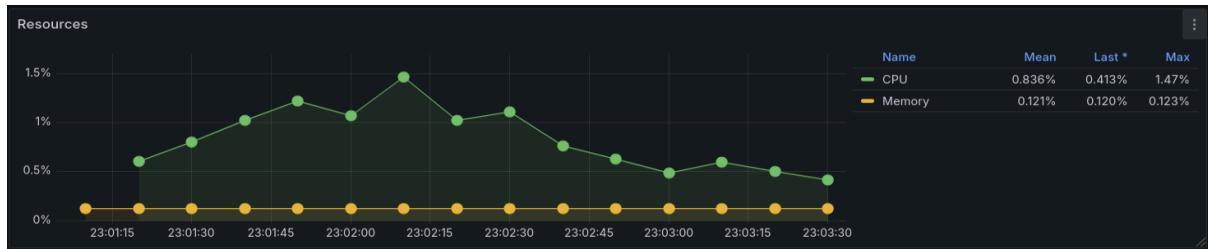
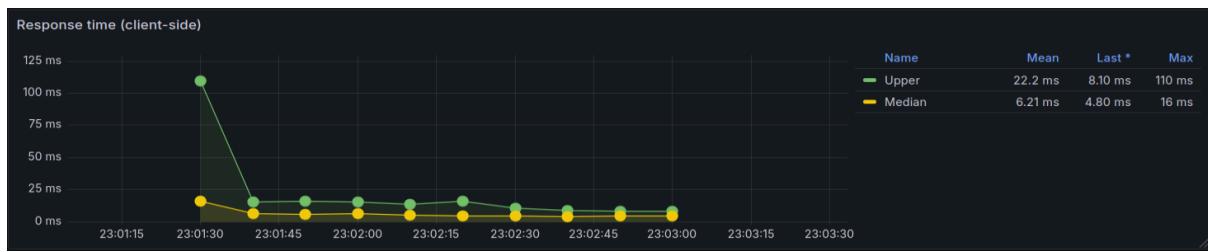
El endpoint actualiza las tasas de cambio mantenidas en memoria y las persiste de forma asíncrona, sin bloqueos por E/S de disco.

Durante la prueba se alcanzó un máximo de 80 req/s y un promedio de 54,5 req/s, sin errores ni interrupciones.

Las latencias fueron bajas y estables (mediana 6,2 ms, pico 110 ms al inicio), con rápida estabilización por debajo de 10 ms.

El consumo de recursos se mantuvo reducido (CPU promedio 0,83 %, memoria 0,12 %), confirmando que las actualizaciones se procesan con eficiencia y sin impacto significativo en el rendimiento general del sistema.





### Escenario: Carga, Endpoint: GET /log

GET /log — degradación por payload no acotado. El endpoint devuelve el log completo; a medida que el archivo crece, aumentan linealmente lectura de disco, serialización y bytes transferidos

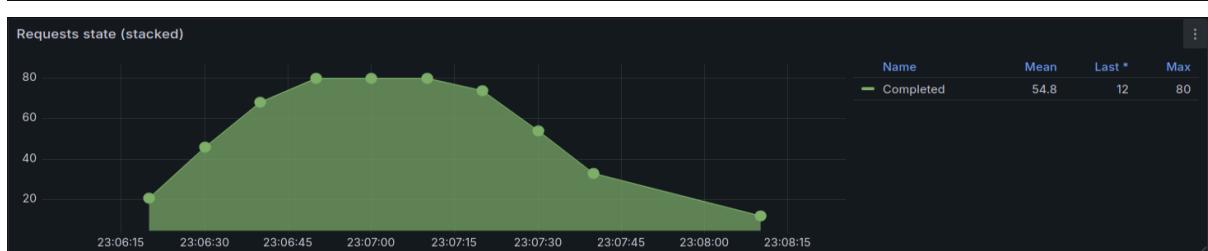
Ejemplo con content-length = 288321 bytes

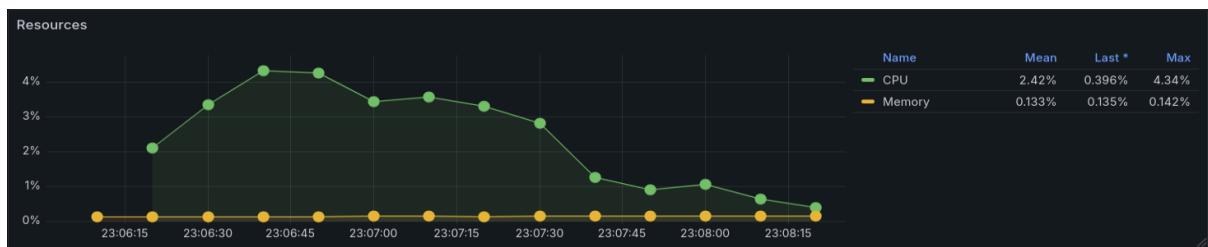
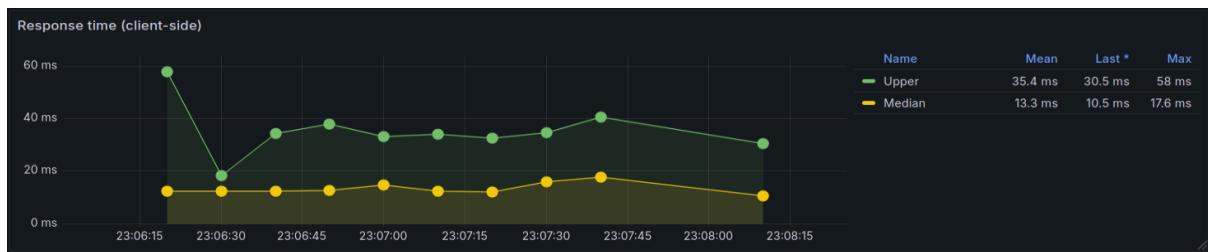
El endpoint *GET /log* devuelve el historial completo de transacciones, lo que genera un payload grande ( $\approx$  288 KB) y un aumento del tiempo de respuesta a medida que el log crece.

Durante la prueba se alcanzó un máximo de 80 req/s y un promedio de 54,8 req/s, sin errores, pero con latencias más altas que en otros endpoints (mediana 13,3 ms, pico 58 ms).

El uso de CPU fue mayor (promedio 2,4 %, pico 4,3 %), mientras que la memoria se mantuvo estable (0,13 %).

Estos resultados muestran que el rendimiento se degrada de forma proporcional al tamaño del log, por lo que sería recomendable implementar paginación o filtros para limitar la cantidad de datos devueltos por cada solicitud.





### Escenario: Carga, Endpoint: POST /exchange

Cada transacción involucra múltiples operaciones internas y una latencia artificial de entre 200 y 400 ms por transferencia, definida en el código para simular el tiempo de respuesta de servicios externos.

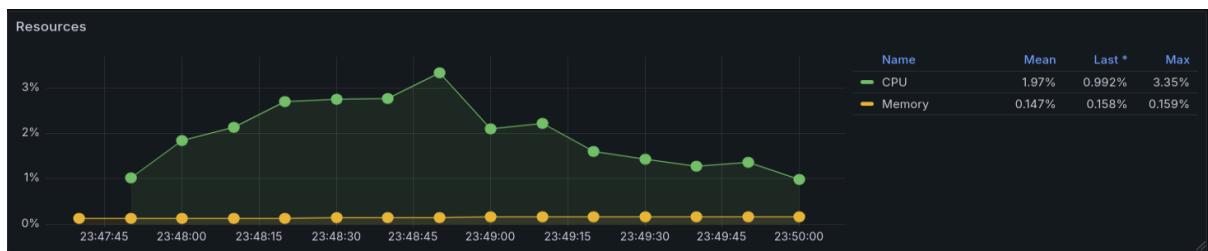
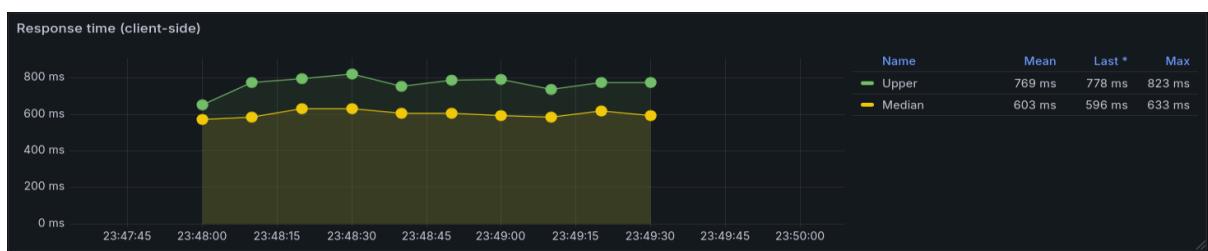
Los gráficos muestran un comportamiento estable, con un máximo de 80 req/s y un promedio de 54,4 req/s, sin errores ni interrupciones.

El tiempo de respuesta es considerablemente mayor que en los demás endpoints, con una mediana de 603 ms, promedio (upper) de 769 ms y un pico de 823 ms, valores coherentes con las demoras del proceso de intercambio.

A pesar de ello, la latencia se mantiene constante, lo que indica que el sistema tolera adecuadamente la carga concurrente sin degradar el rendimiento.

En cuanto al uso de recursos, el consumo promedio fue 1,97 % de CPU (pico 3,35 %) y 0,15 % de memoria, manteniéndose dentro de rangos bajos pese a la complejidad de la operación.

### Gráficos de negocio



## Escenario: Stress, Endpoint: POST /exchange

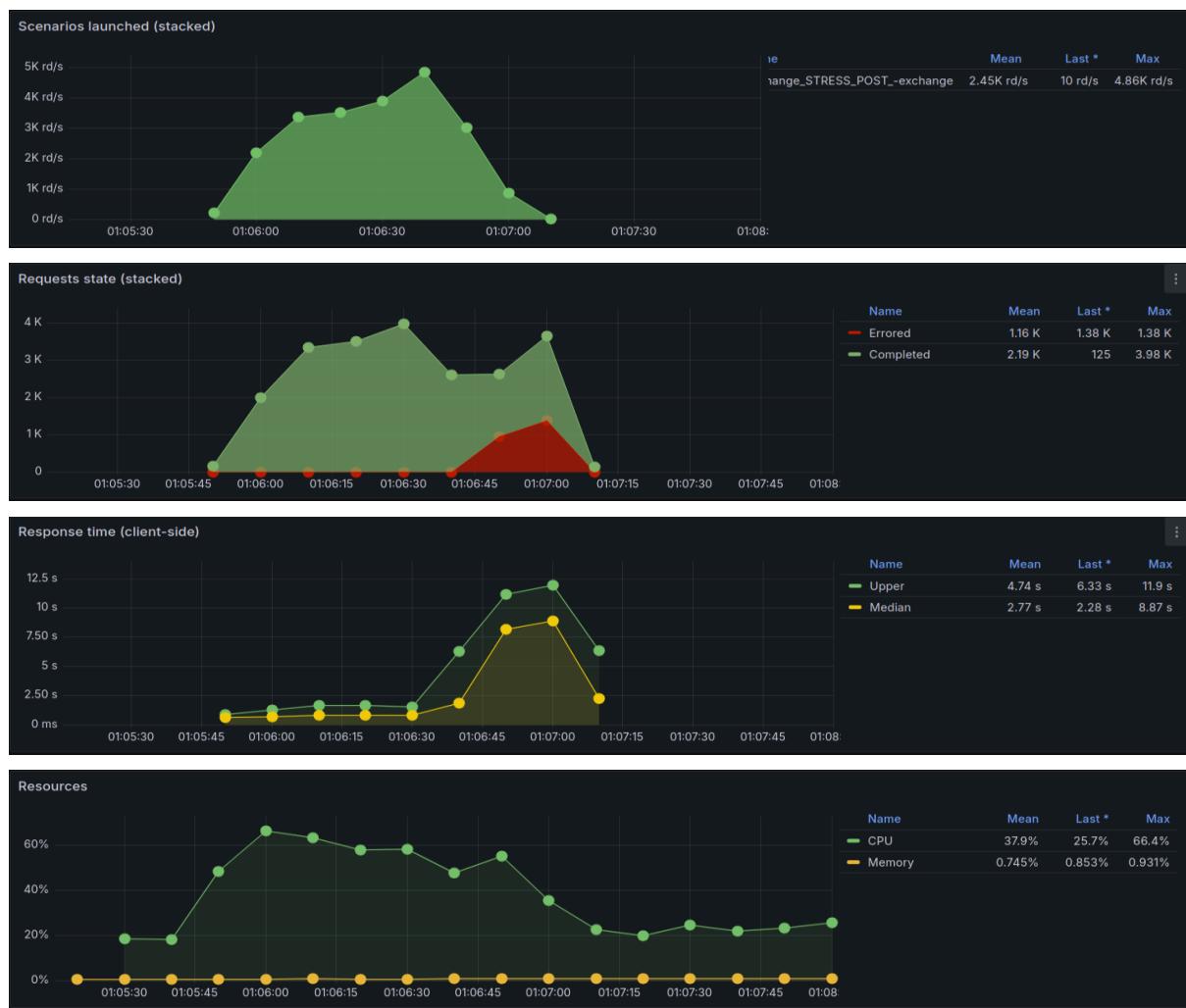
El endpoint *POST /exchange* fue sometido a una carga extrema, alcanzando un pico de 4,8 K req/s y un promedio de 2,4 K req/s.

Bajo estas condiciones, comenzó a registrar errores en torno a un tercio de las solicitudes, evidenciando saturación del servicio.

Los tiempos de respuesta aumentaron significativamente (mediana 2,8 s, pico 11,9 s) debido a la alta concurrencia y a la latencia interna del proceso de intercambio.

El uso de CPU se incrementó hasta un promedio del 38 % (pico 66 %), mientras que la memoria se mantuvo estable ( $\approx 0,75 \%$ ).

En síntesis, el sistema conserva estabilidad estructural, pero no soporta cargas intensas sin degradar rendimiento, marcando el límite operativo de la arquitectura base.



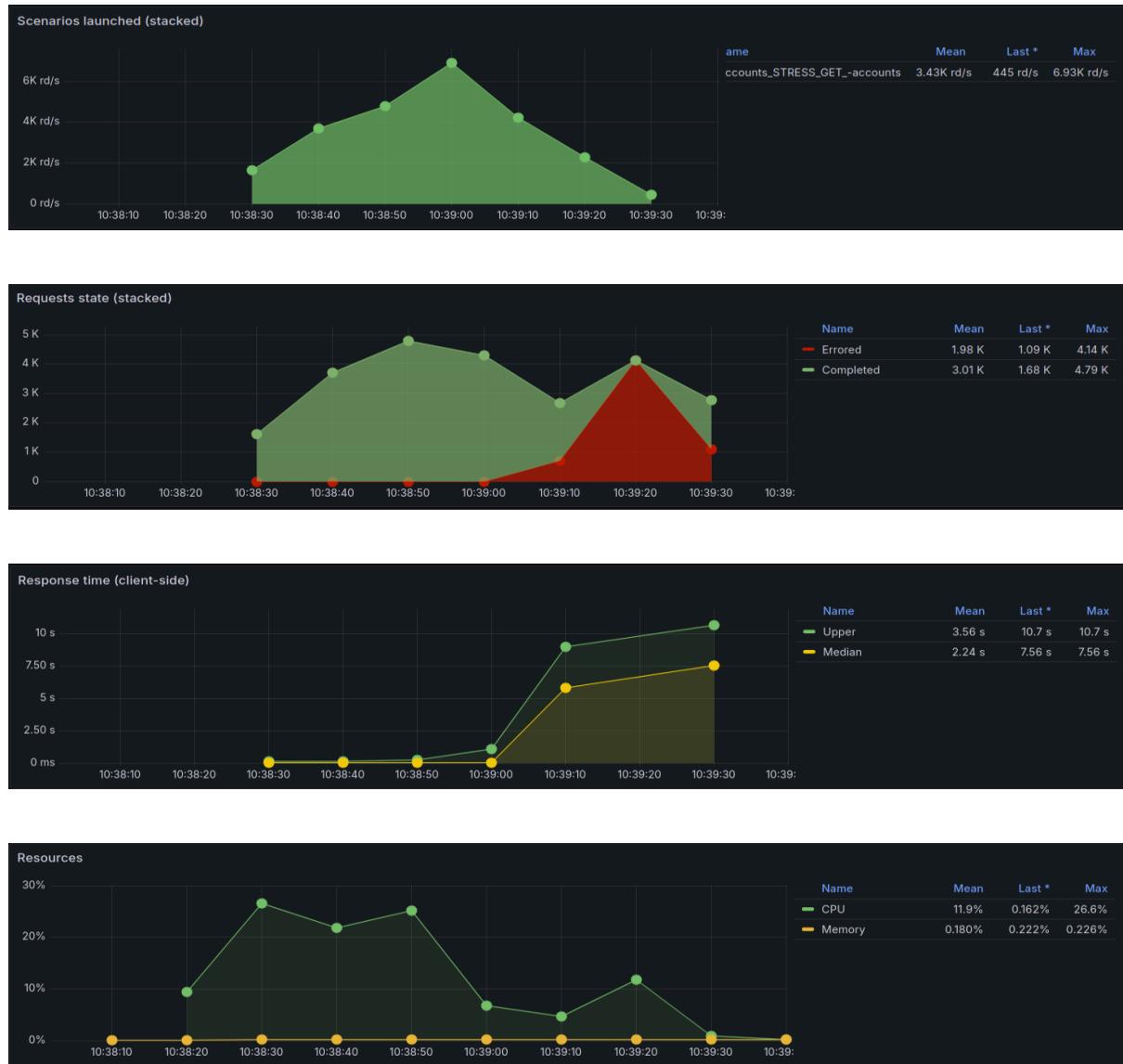
## Escenario: Stress, Endpoint: GET /accounts

Bajo condiciones de estrés, el endpoint *GET /accounts* alcanzó un pico de 6,9 K req/s y un promedio de 3,4 K req/s, comenzando a registrar errores en aproximadamente el 40 % de las solicitudes una vez superado el umbral de carga.

El tiempo de respuesta se incrementó de forma notable (mediana 2,2 s, pico 10,7 s), evidenciando saturación en la atención concurrente de peticiones.

El uso de CPU alcanzó un promedio de 12 % con picos de 26 %, mientras que la memoria permaneció estable en torno al 0,18 %, lo que indica que el cuello de botella se debe principalmente a la carga de procesamiento.

El sistema logra responder correctamente hasta cierto nivel de concurrencia, pero bajo cargas extremas experimenta degradación de rendimiento y errores crecientes, marcando el límite de escalabilidad de la arquitectura base.



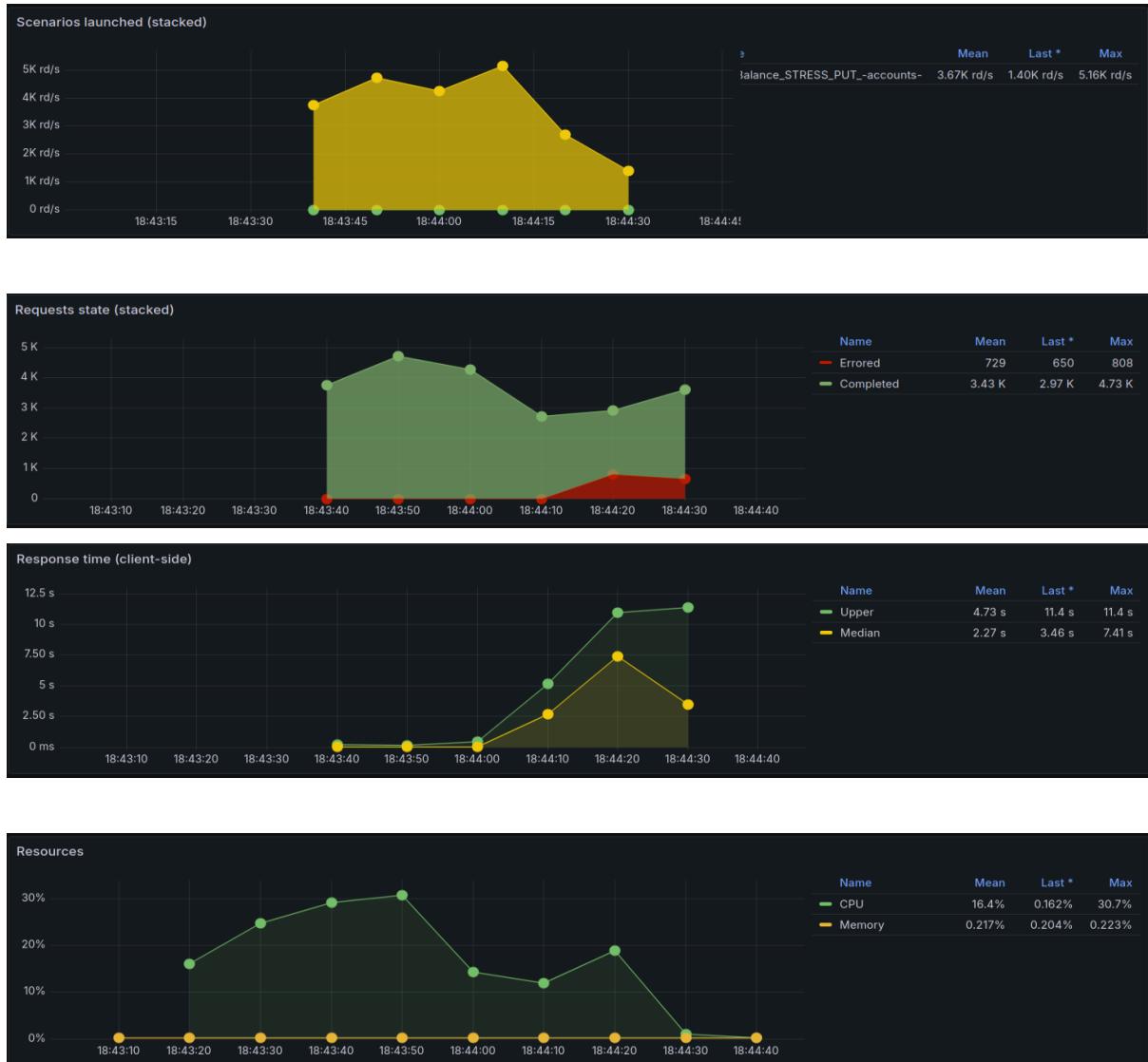
### Escenario: Stress, Endpoint: PUT /accounts/{id}/balance

En condiciones de alta concurrencia, el endpoint alcanzó un pico de 5,1 K req/s y un promedio de 3,7 K req/s, comenzando a registrar errores en torno al 15–20 % de las solicitudes.

El tiempo de respuesta aumentó progresivamente (mediana 2,3 s, pico 11,4 s), mostrando la dificultad del sistema para sostener operaciones de escritura simultáneas.

El uso de CPU se elevó a un promedio de 16 %, con picos cercanos al 31 %, mientras que la memoria permaneció estable en ≈ 0,22 %, indicando que el límite de rendimiento proviene de la carga de procesamiento y no del consumo de memoria.

En síntesis, el endpoint mantiene funcionalidad bajo carga intensa, pero satura rápidamente ante escrituras concurrentes, evidenciando la falta de mecanismos de concurrencia y sincronización en la arquitectura base.



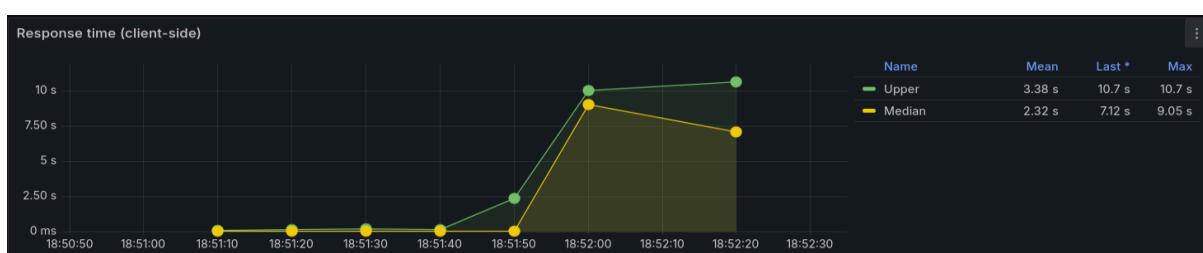
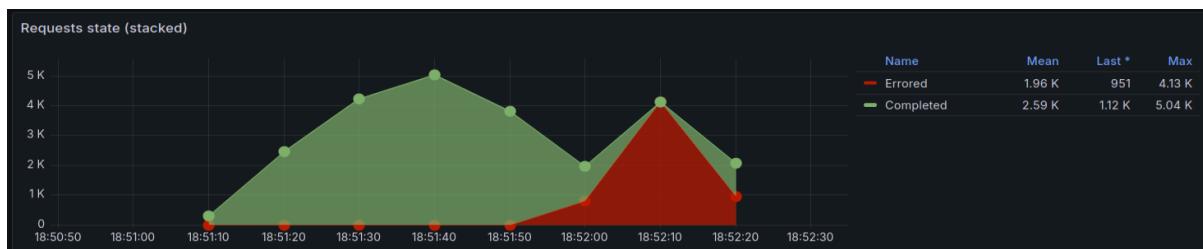
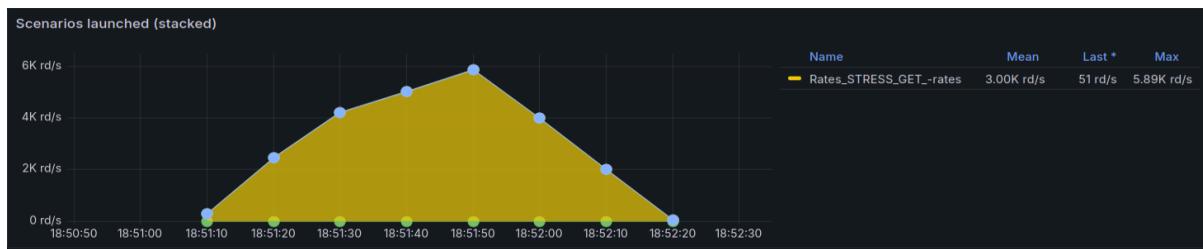
### Escenario: Stress, Endpoint: GET /rates

Durante la prueba de estrés, el endpoint alcanzó un pico de 5,9 K req/s y un promedio de 3,0 K req/s, comenzando a registrar errores en torno al 25–30 % de las solicitudes una vez superado el umbral de carga.

El tiempo de respuesta se degradó progresivamente (mediana 2,3 s, pico 10,7 s), reflejando la saturación del servicio ante múltiples accesos simultáneos.

El uso de CPU aumentó hasta un promedio de 11 % (pico 25 %), mientras que la memoria se mantuvo estable en ≈ 0,22 %, lo que indica que el cuello de botella proviene del procesamiento concurrente y no del consumo de memoria.

En conclusión, el endpoint mantiene estabilidad estructural, pero pierde eficiencia bajo alta concurrencia, mostrando un punto claro de saturación a partir del cual la latencia y los errores crecen rápidamente.

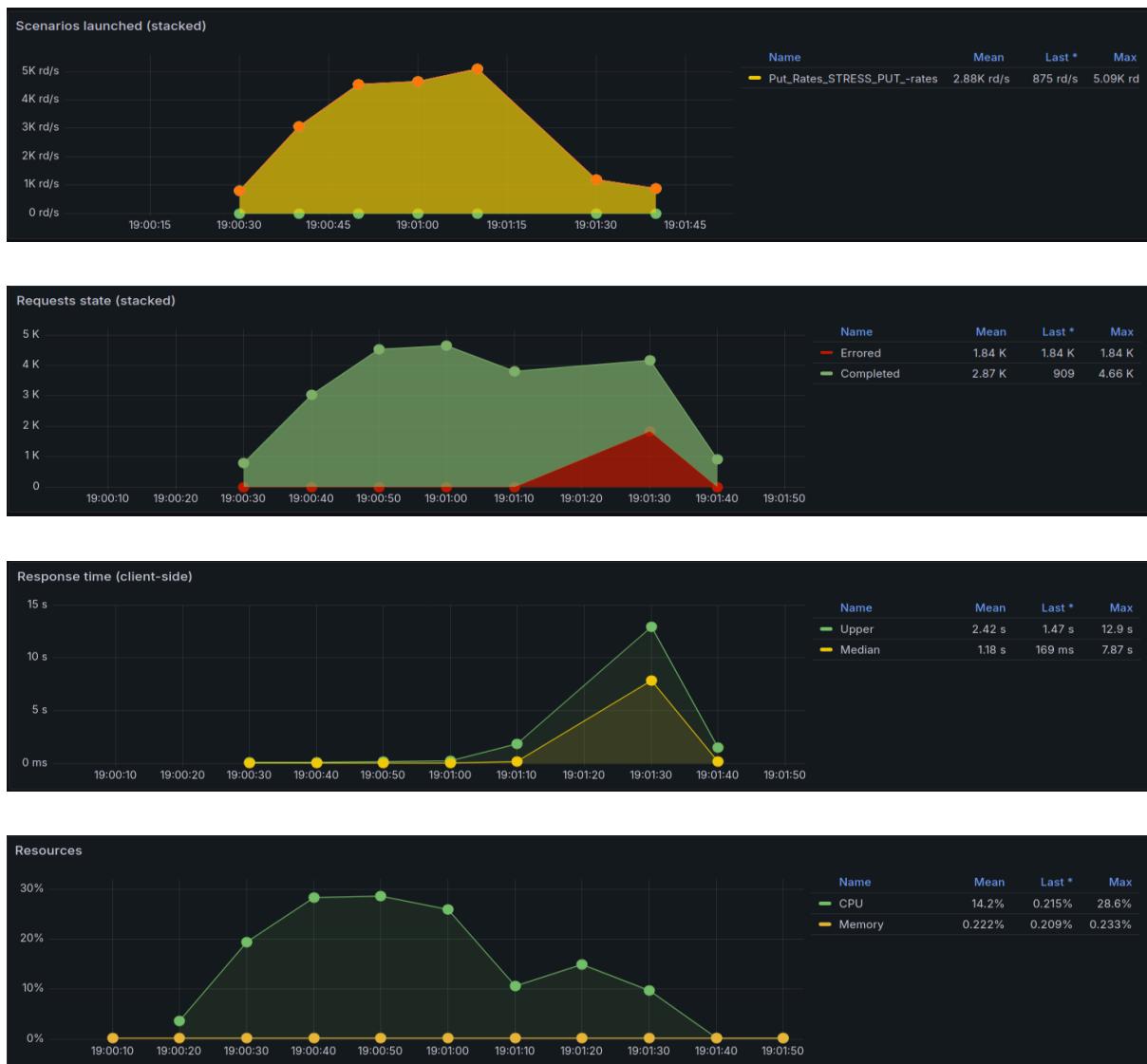


## Escenario: Stress, Endpoint: PUT /rates

Bajo carga extrema, el endpoint alcanzó un **pico de 5,0 K req/s** y un **promedio de 2,9 K req/s**, comenzando a registrar **errores en torno al 35–40 % de las solicitudes** al superar el umbral de concurrencia.

El **tiempo de respuesta** se incrementó notablemente (**mediana 1,2 s, pico 12,9 s**), evidenciando saturación en las operaciones de escritura simultánea sobre las tasas en memoria.

El **uso de CPU** promedió **14 %** con **picos cercanos al 29 %**, mientras que la **memoria** se mantuvo estable alrededor del **0,22 %**, lo que indica que la degradación proviene del procesamiento concurrente y no del almacenamiento.



## 6.2 Rate Limiting

La incorporación del **Rate Limiting** representó una mejora estructural en la arquitectura del sistema, orientada a fortalecer su **resiliencia, disponibilidad y estabilidad** frente a escenarios de alta concurrencia.

La táctica se implementó mediante la librería *express-rate-limit*, aplicando un **enfoque multicapa** con tres niveles de protección diferenciados:

- un **limitador global** (1000 req/15 min por IP) para mitigar abuso generalizado,
- un **limitador estricto** (10 req/min) para operaciones críticas de escritura (POST/PUT),
- y un **limitador de consulta** (60 req/min) para endpoints de lectura (GET).

Cada componente fue instrumentado con métricas enviadas a **StatsD** y visualizadas en **Grafana**, permitiendo un monitoreo continuo de desempeño y comportamiento del tráfico.

Desde una perspectiva de **atributos de calidad**, el rate limiting demostró mejoras claras en **disponibilidad**, al prevenir la saturación y mantener la API operativa incluso bajo picos de demanda; en **resiliencia**, al aislar los efectos de la sobrecarga sin comprometer el servicio completo; y en **previsibilidad**, al ofrecer tiempos de respuesta consistentes y un consumo de recursos controlado.

Los resultados experimentales mostraron reducciones sustanciales de CPU (en promedio un 50 % menos respecto al caso base bajo estrés), ausencia de degradación progresiva, y una respuesta estable en todos los endpoints, incluso en condiciones de carga extrema.

No obstante, esta táctica implica **trade-offs**: el **throughput efectivo** se reduce, ya que parte del tráfico es descartado o pospuesto al alcanzar los límites definidos. Este sacrificio es deliberado y busca priorizar la **continuidad del servicio** sobre el volumen procesado, garantizando que las solicitudes válidas sean atendidas con tiempos razonables.

Como conclusión, la aplicación del **Rate Limiting multicapa** reforzó la robustez de la arquitectura base al introducir mecanismos de defensa adaptativos frente a cargas imprevistas, evitando efectos cascada y asegurando una **degradación controlada** en lugar de un colapso total del sistema. Combinada con el monitoreo continuo y métricas centralizadas, esta táctica sienta las bases para un modelo de **escalabilidad sostenible**, permitiendo que el sistema mantenga su calidad de servicio incluso ante patrones de uso variables o tráfico anómalo.

#### Escenario: Carga, Endpoint: GET /accounts

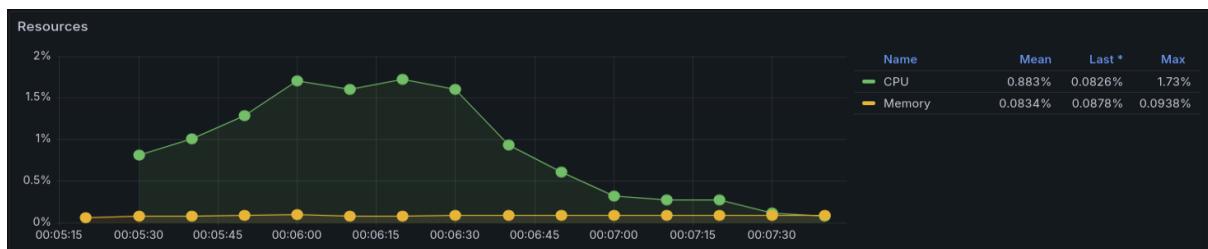
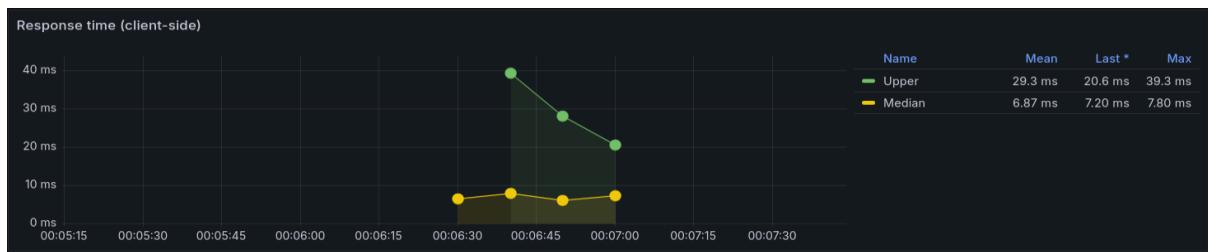


#### Escenario: Carga, Endpoint: PUT /accounts/{id}/balance

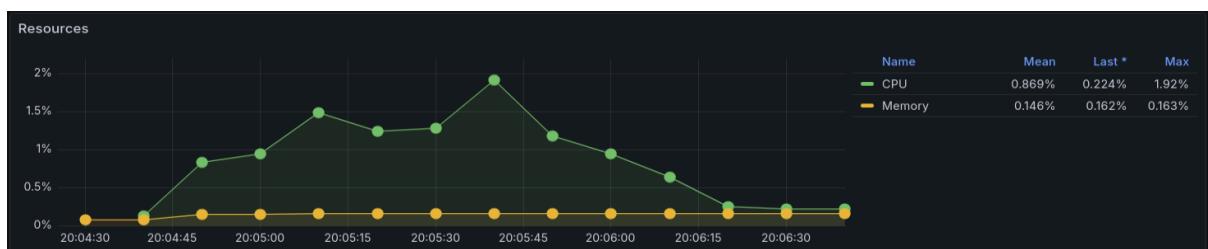
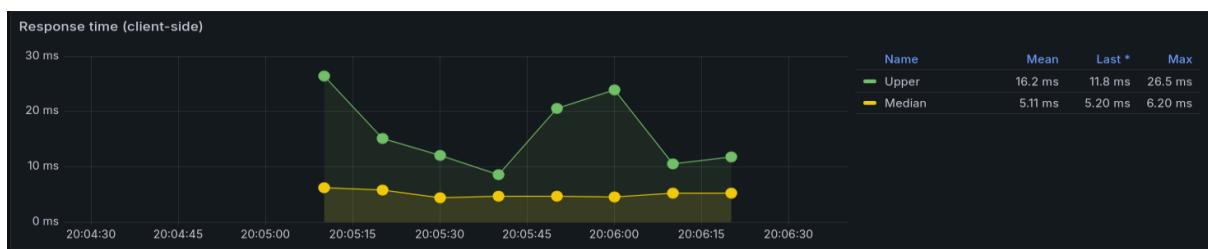
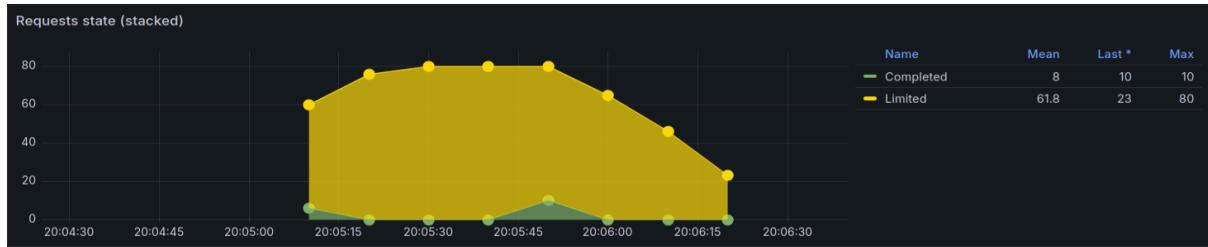
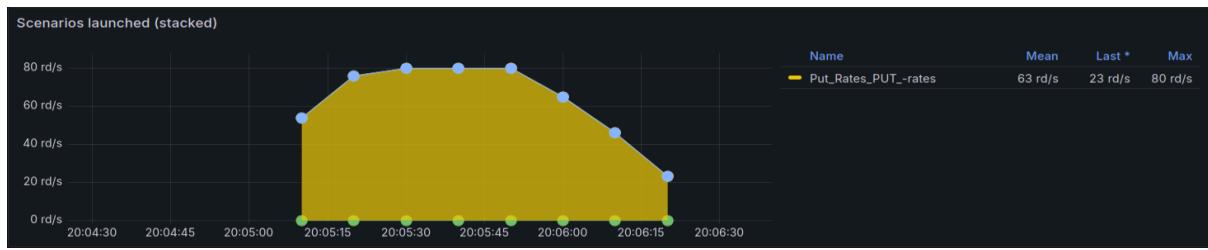


## Escenario: Carga, Endpoint: GET /rates





## Escenario: Carga, Endpoint: PUT /rates

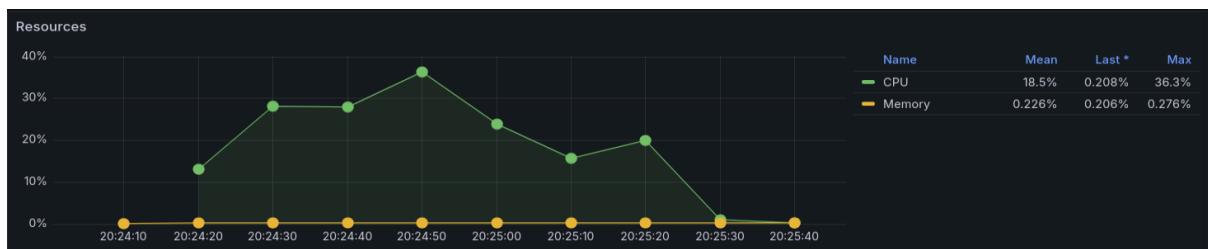
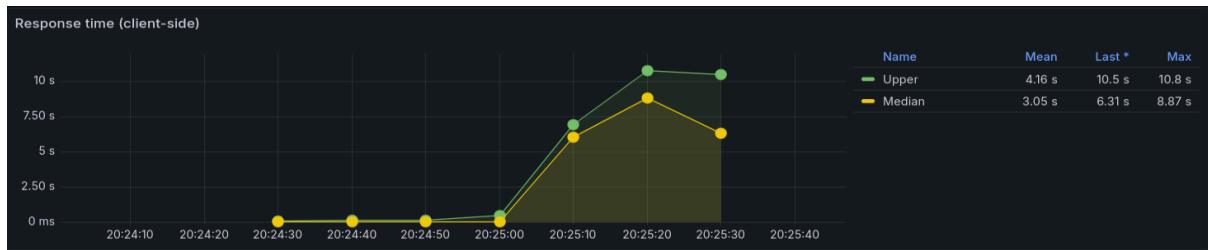


## Escenario: Carga, Endpoint: POST /exchange

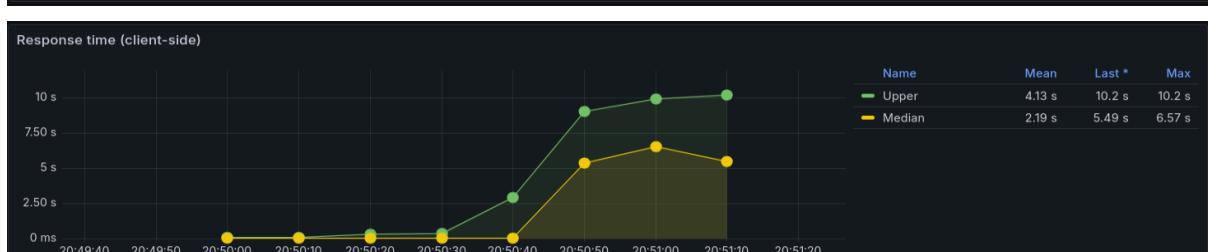
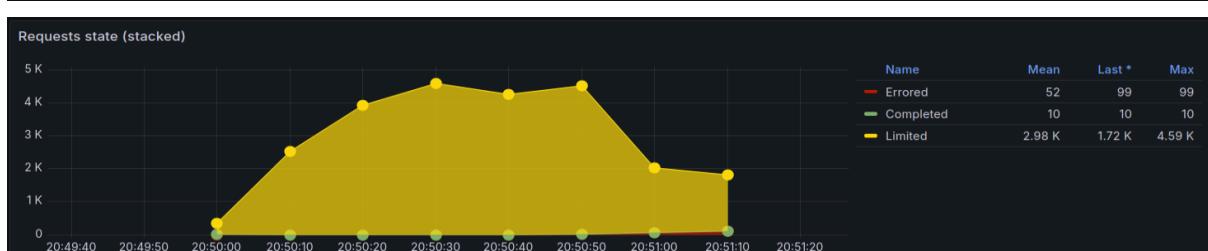
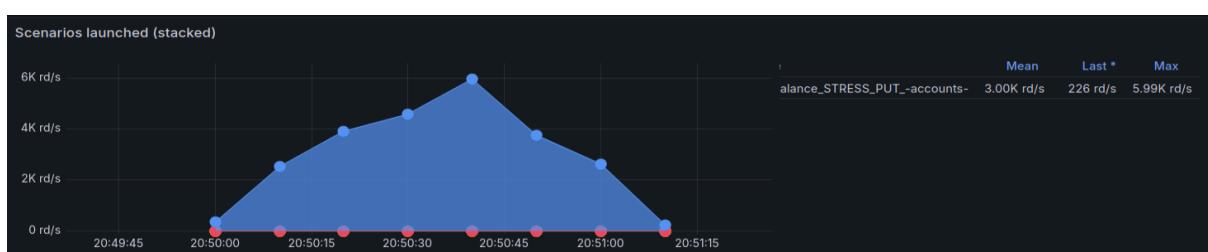


## Escenario: Stress, Endpoint: GET /accounts





### Escenario: Stress, Endpoint: PUT /accounts/{id}/balance

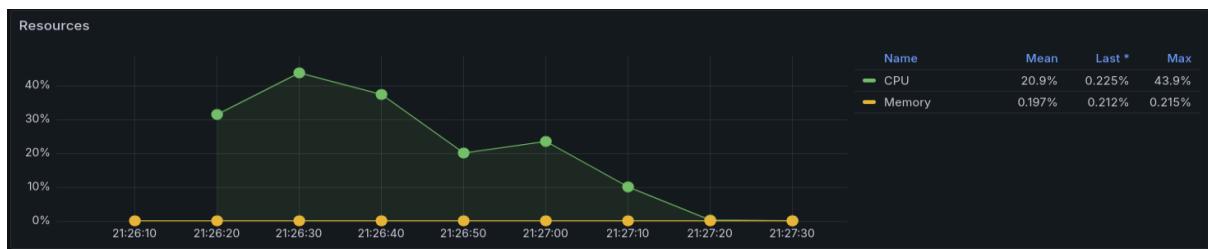
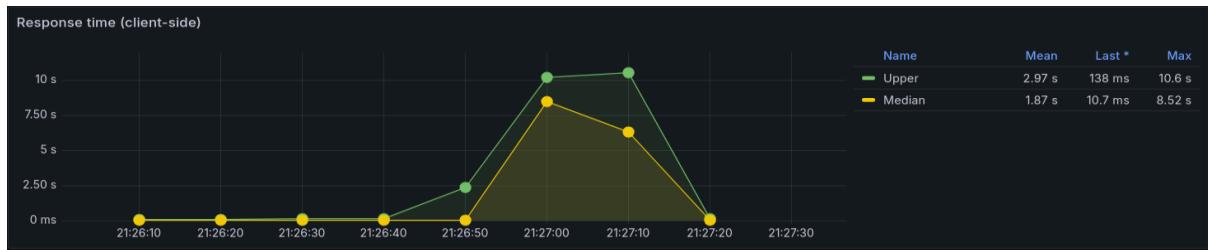


## Escenario: Stress, Endpoint: GET /rates

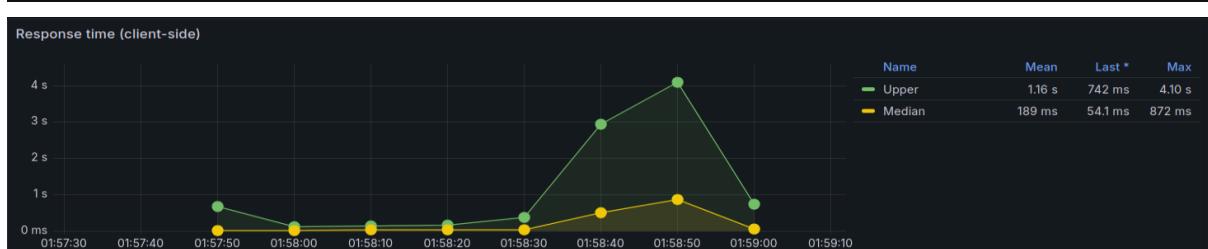
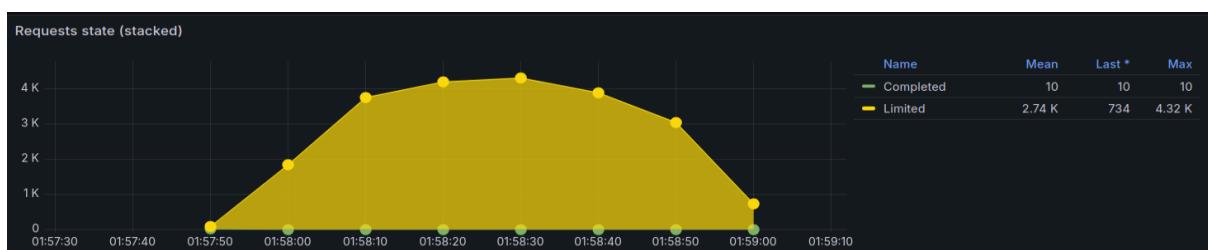
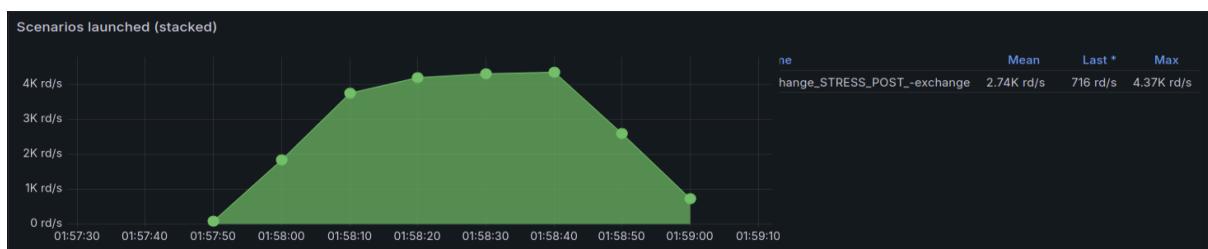


## Escenario: Stress, Endpoint: PUT /rates.





### Escenario: Stress, Endpoint: POST /exchange



## 6.3 Replicación

Para el escenario de replicación, se replica el nodo existente de la aplicación en el caso base, generando 2 réplicas adicionales idénticas a la original. Se configura Nginx de tal manera que tome el

rol de load balancer, distribuyendo las solicitudes entre los 3 posibles nodos. Como configuración por default, Nginx distribuye las solicitudes a partir del algoritmo Round Robin, es decir las solicitudes se distribuyen de manera equitativa entre las 3 réplicas.

A continuación, se encuentran los escenarios de carga de los distintos endpoints

## Escenarios de Carga

Algo interesante que se logra observar sobre los escenarios de carga con los 3 nodos respecto a los del caso base, es que no se ve una mejora clara respecto al response time, pero lógicamente si se logra ver una mejor optimización del uso de recursos con respecto a el nodo en particular, ya que claramente cada nodo tiene menos trabajo que hacer.

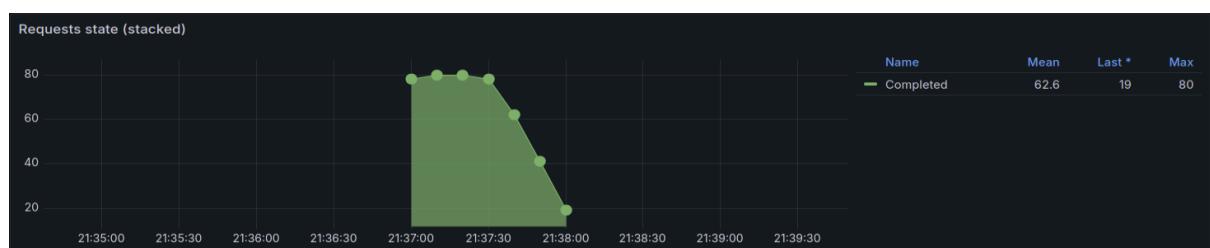
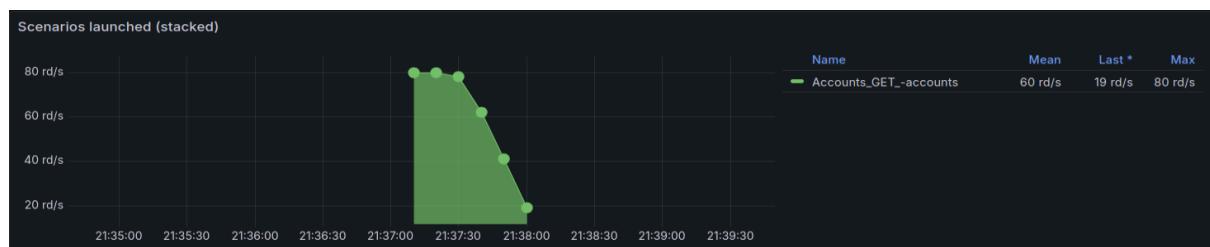
Si nos ponemos a debatir el hecho de que no haya una mejora en la response time, esto podría pasar por 2 factores:

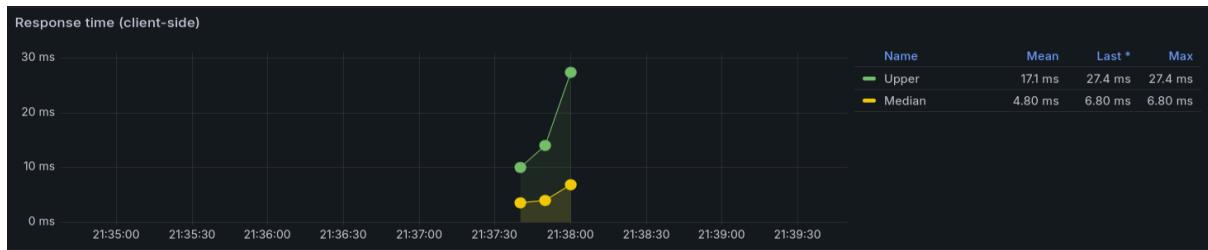
1. El algoritmo de balanceo (round robin) podría ser inapropiado para esta situación ya que puede estar enviando solicitudes a nodos que ya están sobrecargados. PERO, también podemos ver por los gráficos que los nodos tienen aproximadamente el mismo uso de memoria y CPU, por lo que están correctamente balanceados. Entonces nos inclinamos por la opción 2:
2. Hay una falla en la sincronización o latencia en la comunicación. Esto puede deberse a que como las solicitudes son simples, se demora más tiempo en distribuir las solicitudes que en ejecutarse la solicitud

### **Escenario: Carga, Endpoint: GET /accounts**

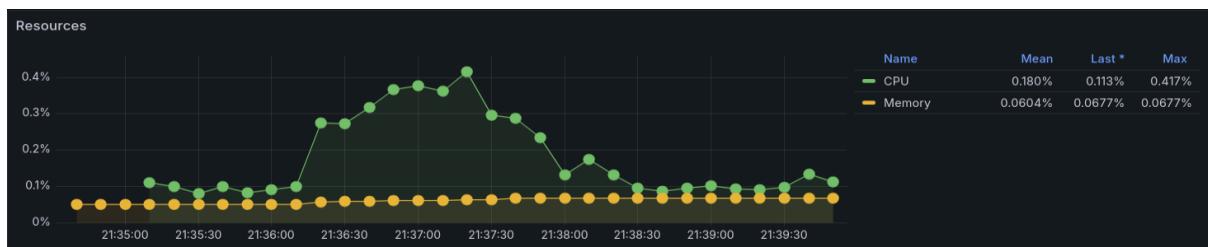
En este caso el response time es de 4,8 ms, mientras que en el caso base es 4,59 ms.

Mientras que el uso de CPU en cada nodo es aproximadamente del 0,173% y memoria 0,07%, mientras que en el caso base es de 0,78% y 0,117%. Se demuestra la observación mencionada previamente. El response time es similar, pero el uso de CPU y Memoria es mucho menor.

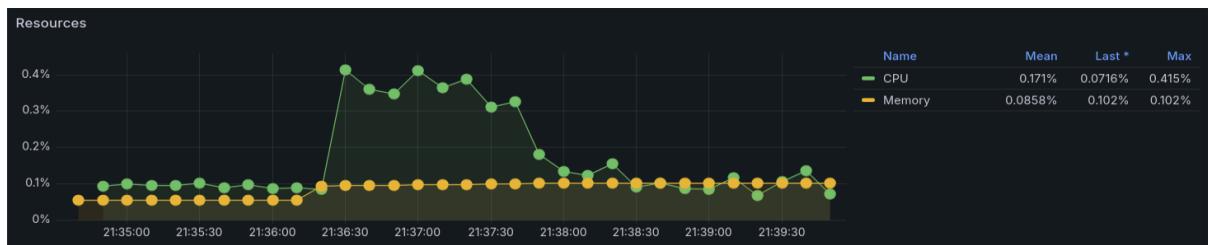




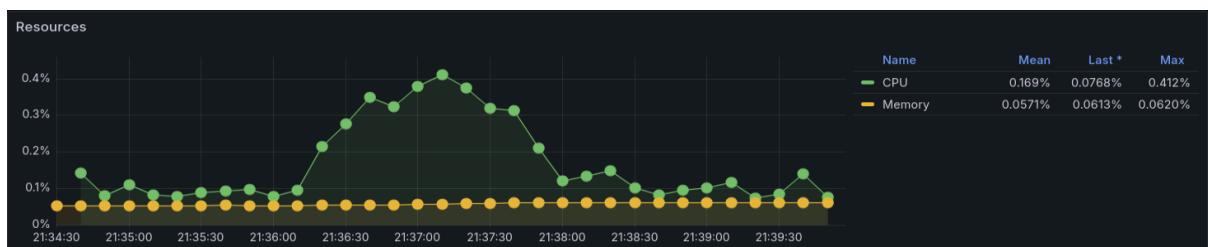
## NODO 1



## NODO 2



## NODO 3

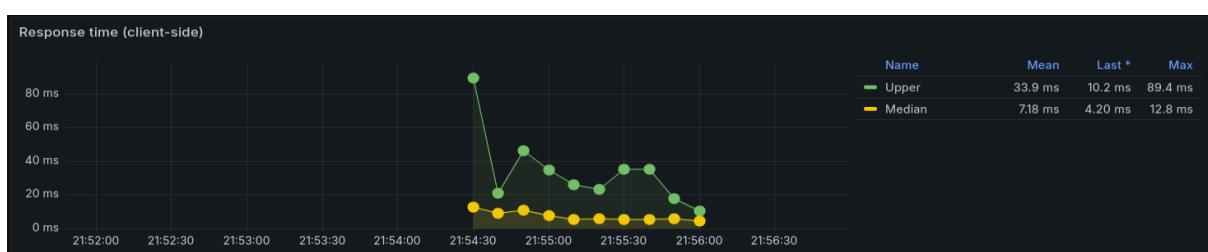
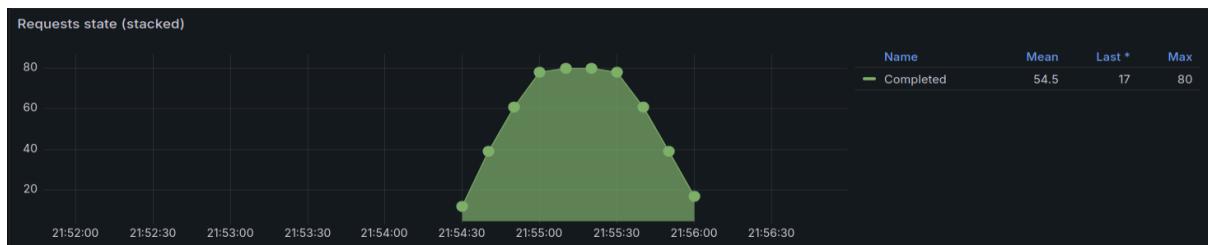


**Escenario: Carga, Endpoint: PUT /accounts/{id}/balance**

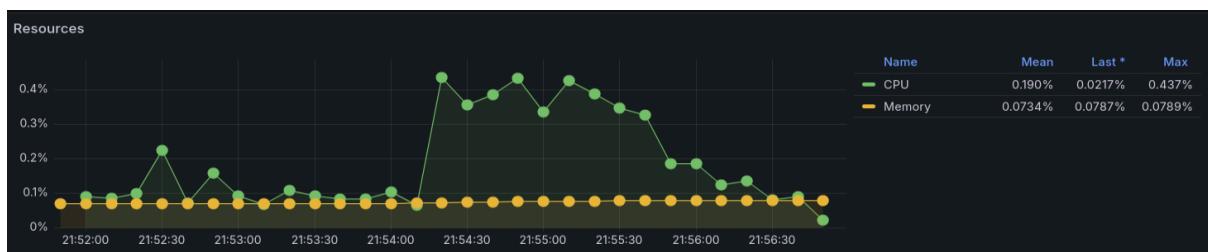
En esta caso las comparaciones son las siguientes:

Response time = 7,18ms vs caso base = 5,31ms.

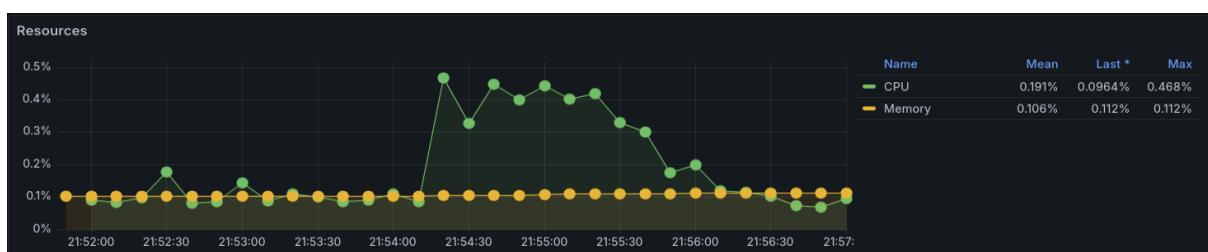
Mientras que uso de CPU=0,19% vs caso base 0,8% (4 veces más en caso base)



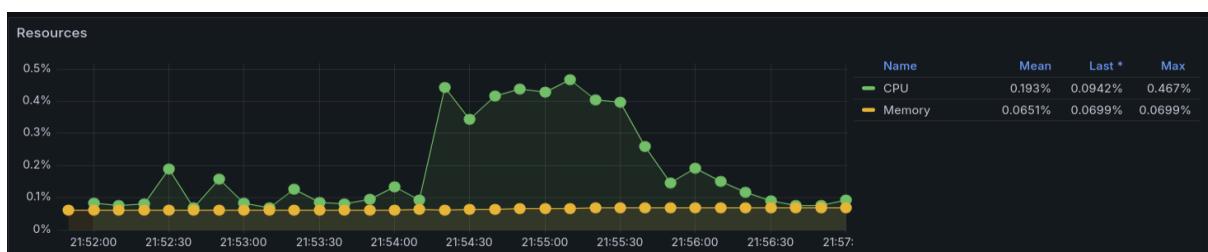
## Nodo 1



## Nodo 2



## Nodo 3



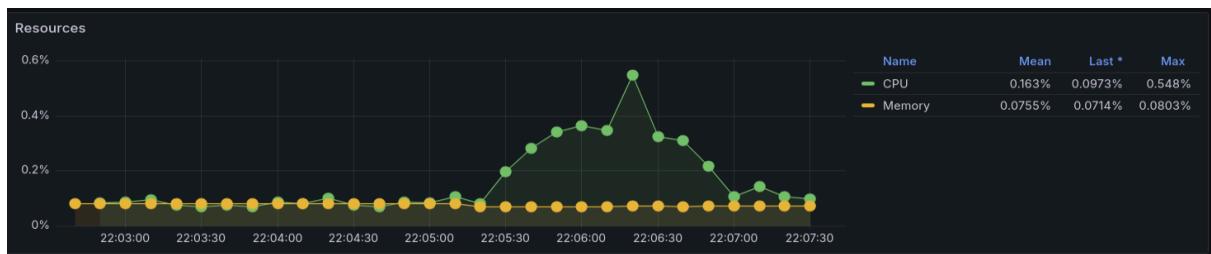
## Escenario: Carga, Endpoint: GET /rates

Response time = 6,1ms vs caso base = 4,47ms

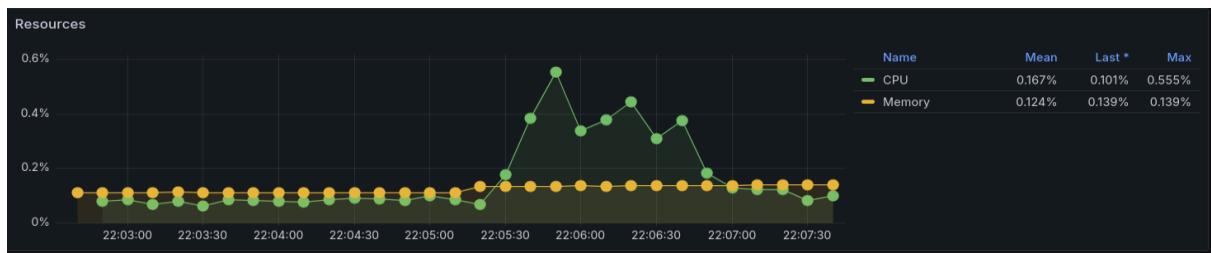
CPU por nodo = 0,16% vs caso base = 0,84% (5 veces más)



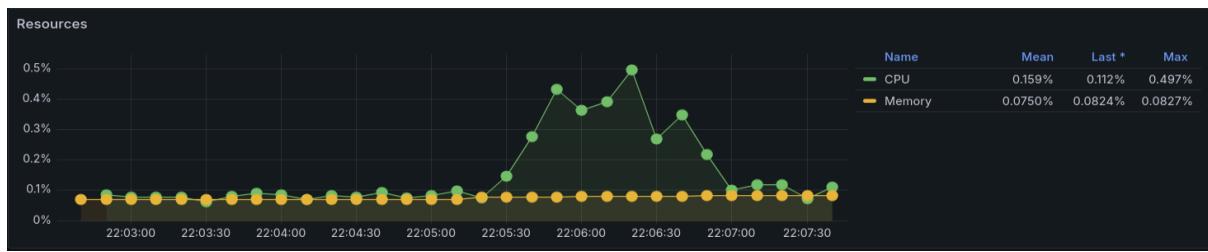
## Nodo 1



## Nodo 2



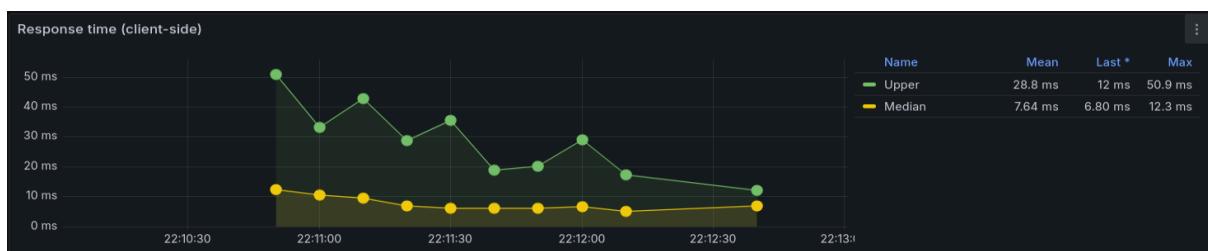
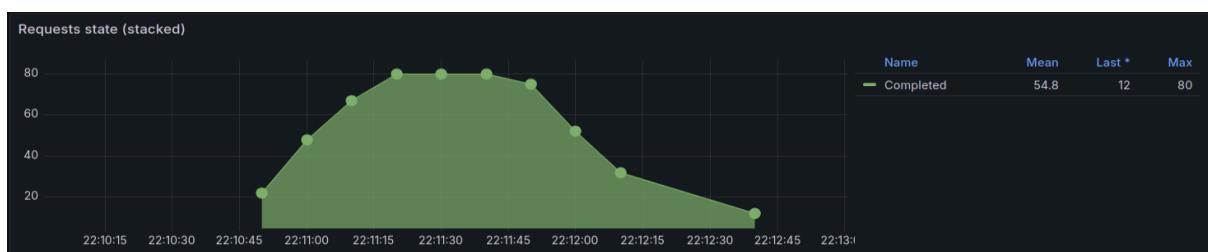
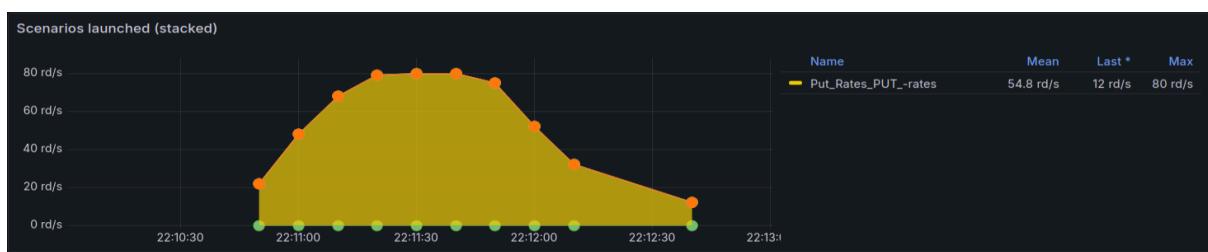
## Nodo 3



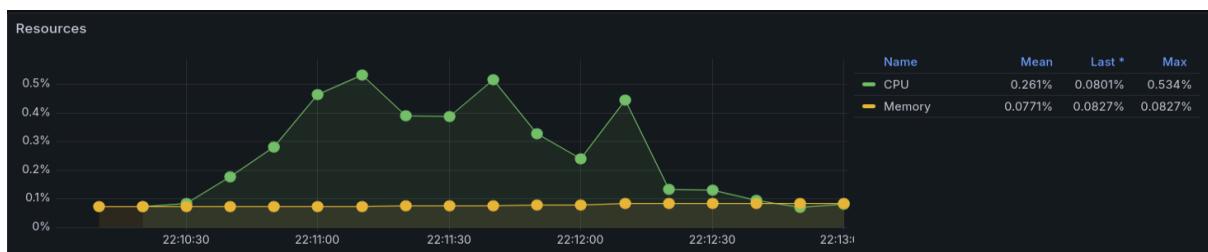
### Escenario: Carga, Endpoint: PUT /rates

Response time = 7,64ms vs caso base = 6,21 ms

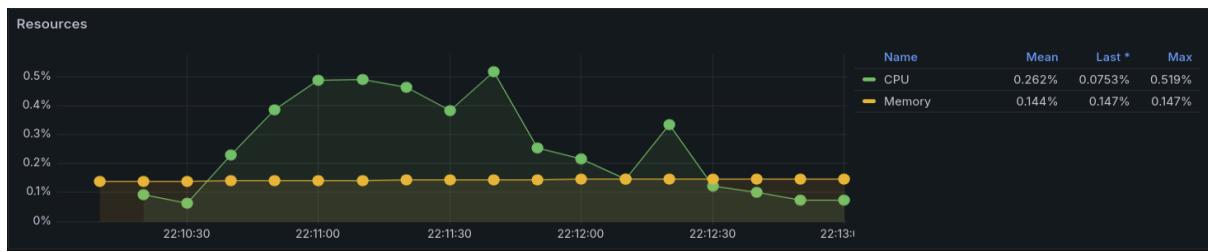
CPU = 0,26% vs caso base = 0,84% (3 veces más, no hay diferencia total)



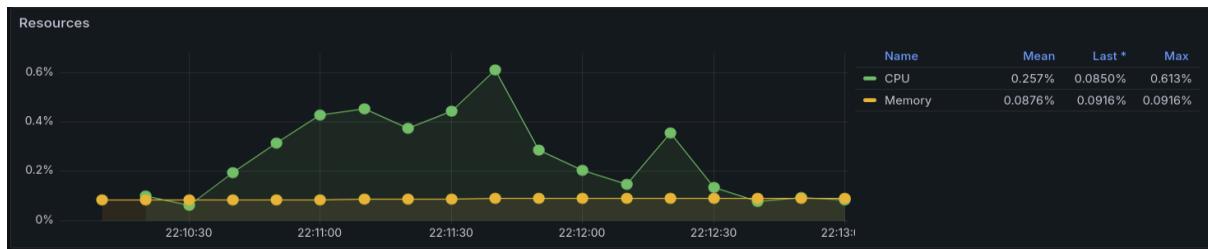
### Nodo 1



### Nodo 2



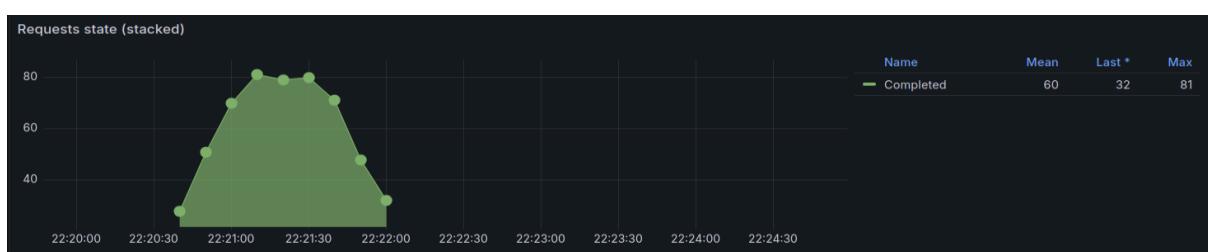
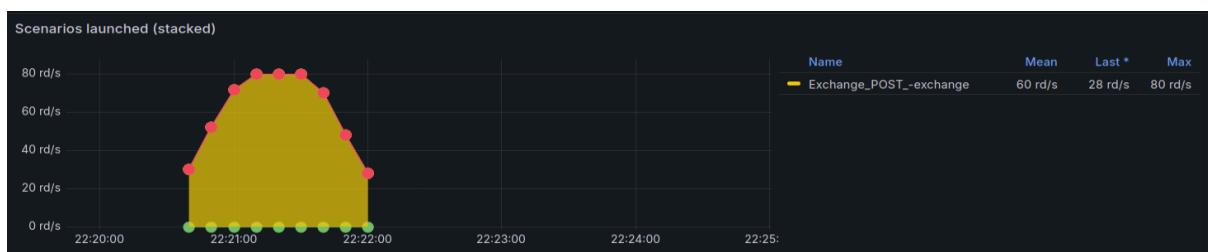
## Nodo 3



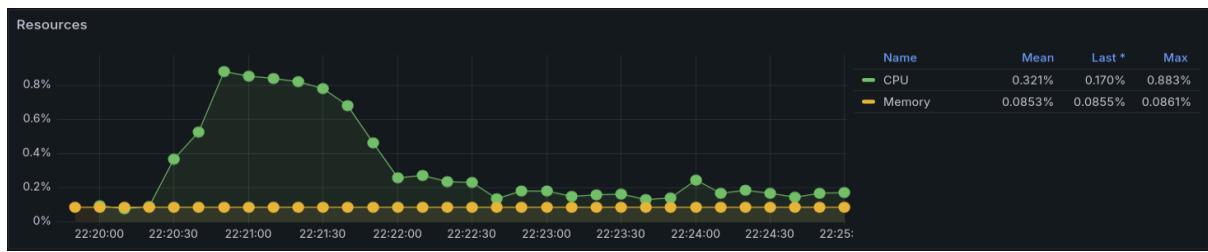
## Escenario: Carga, Endpoint: POST /exchange

Response time = 611 ms vs caso base = 603 ms

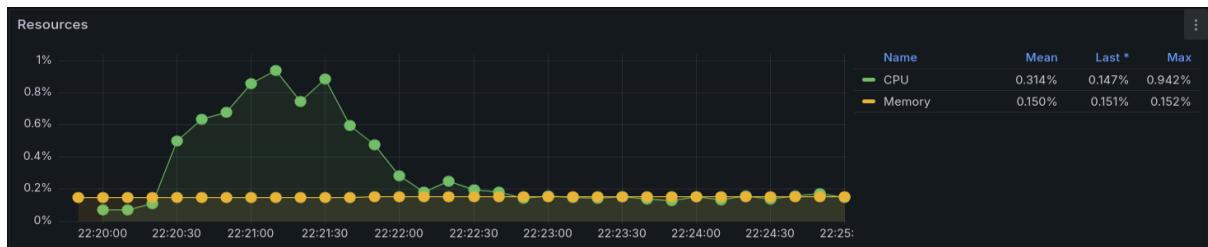
CPU = 0,32% vs caso base = 1,97% (6 veces más)



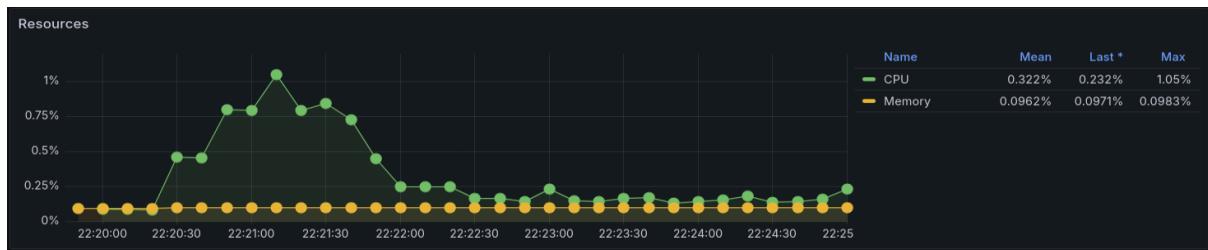
## Nodo 1



## Nodo 2



## Nodo 3



## Escenarios de Stress

A continuación se muestran los distintos escenarios de estrés para cada endpoint en la situación de replicación.

A partir de los distintos valores obtenidos en los escenarios, se puede visualizar que con la arquitectura de replicación, el sistema reacciona mucho mejor en casos de estrés.

Primero y principal, el sistema casi no obtuvo fallos y pudo procesar prácticamente todas las solicitudes obtenidas, aunque sigue obteniendo fallas.

Luego, se redujo el response time en todas los escenarios, lo que muestra que el algoritmo round robin es efectivo en casos de estrés

Finalmente, con respecto al uso de CPU, si sumamos la CPU de los 3 nodos, esta supera al uso del caso base, pero también se puede suponer esto ya que en el caso base hay menos solicitudes procesadas

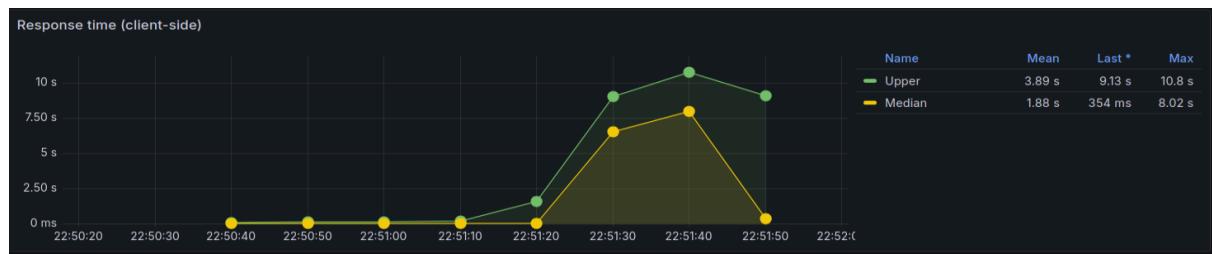
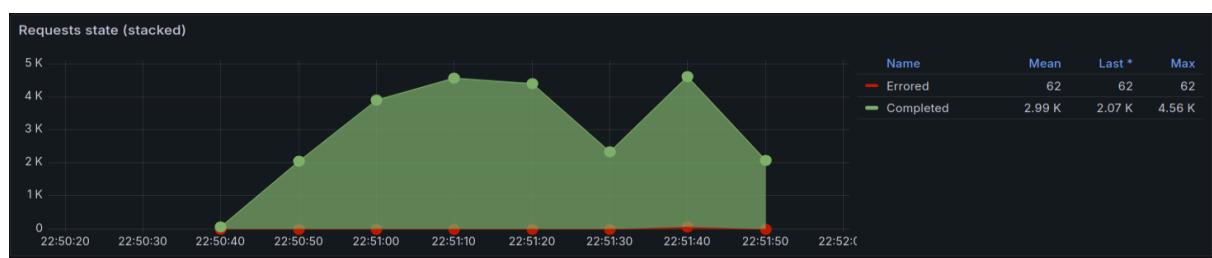
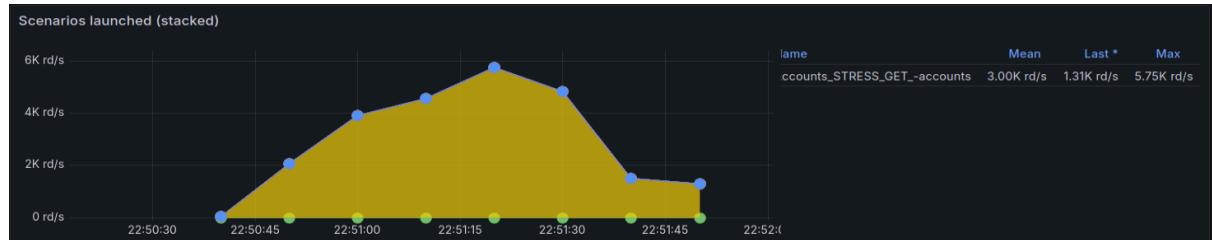
En líneas generales, se podría afirmar que el algoritmo de replicación muestra mejoras de Performance ante situaciones de estrés, pero presenta un trade off del mismo cuando hay bajas solicitudes, ya que hace que se demore más el tiempo total de respuesta a todas las solicitudes.

También muestra mejoras de resiliencia, ya que se redujo en gran medida la cantidad de fallas en los casos de estrés.

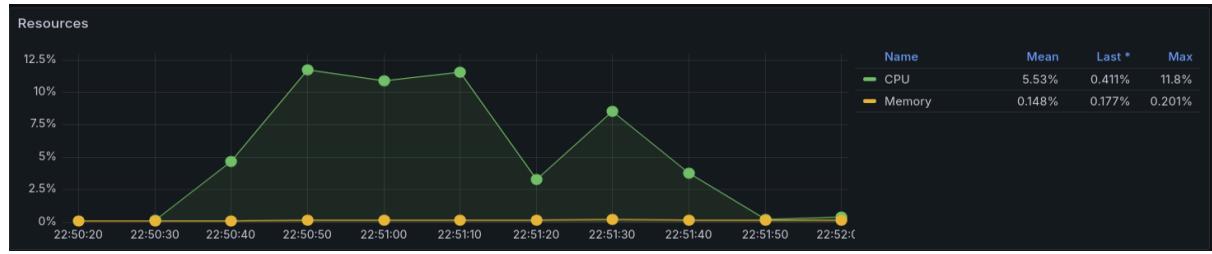
### Escenario: Stress, Endpoint: GET /accounts

Response time = 1,88s vs caso base = 2,24 s

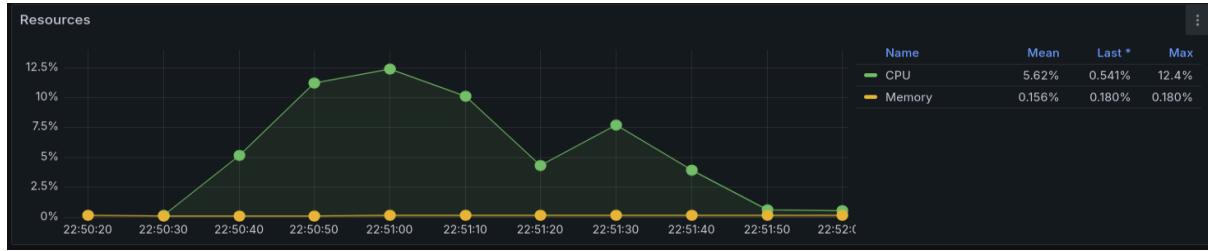
Uso de CPU por nodo = 5,58% vs caso base = 11,9% (menos uso sumando los 3 nodos, pero con errores)



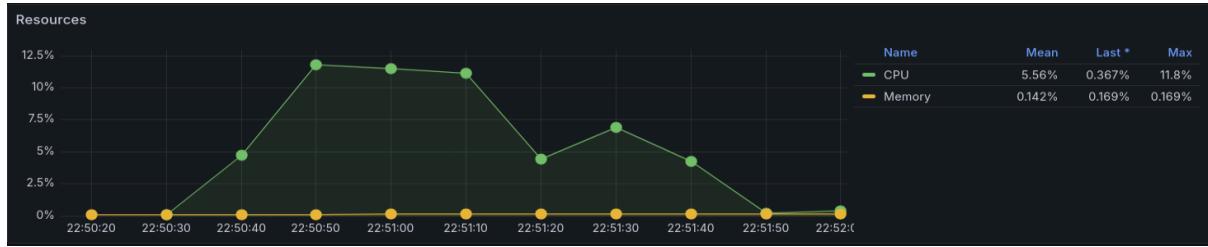
### Nodo 1



### Nodo 2



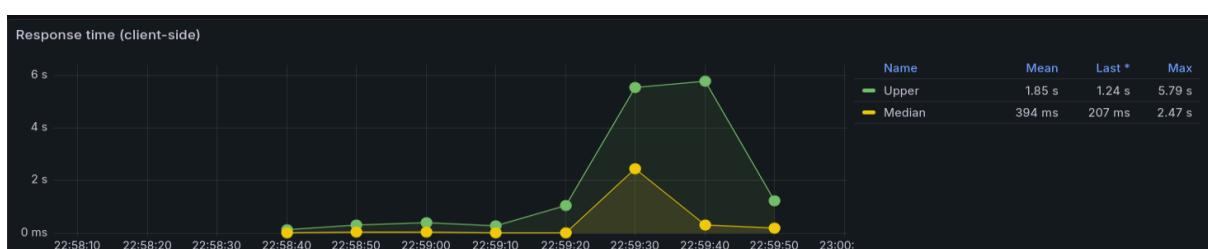
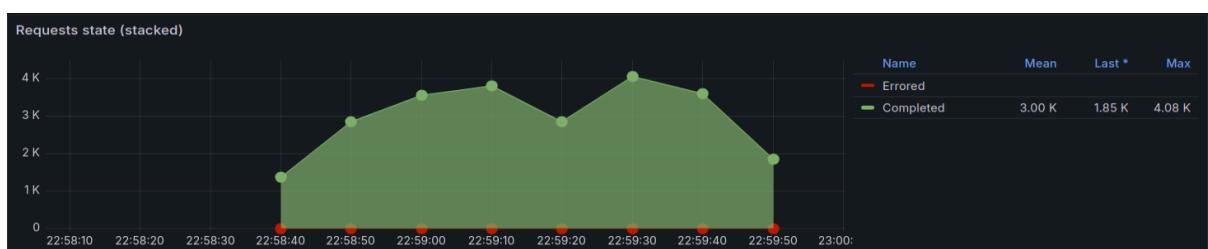
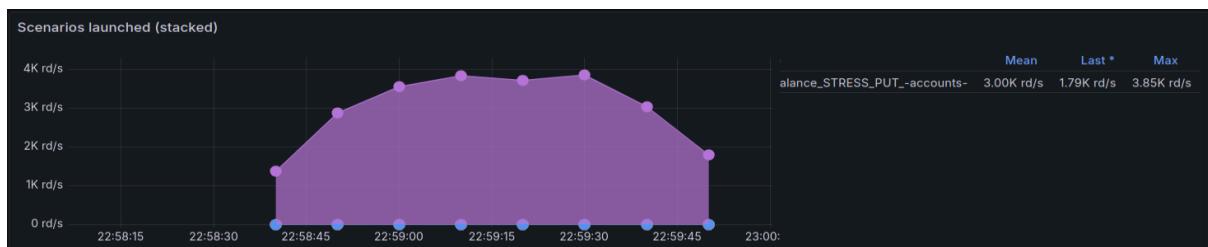
## Nodo 3



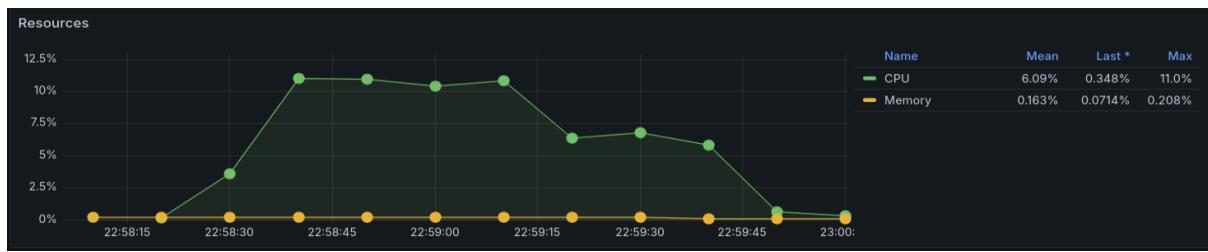
**Escenario: Stress, Endpoint: PUT /accounts/{id}/balance**

Response time = 394ms vs caso base = 2,27 s

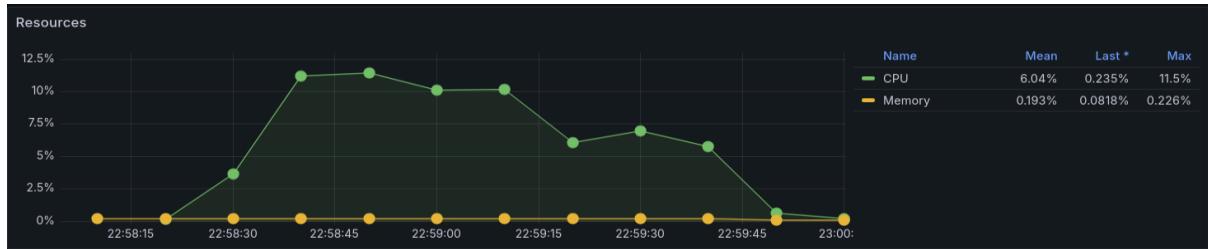
Uso de CPU por nodo = 6% vs caso base = 16,4% (menos uso sumando los 3 nodos, pero con errores)



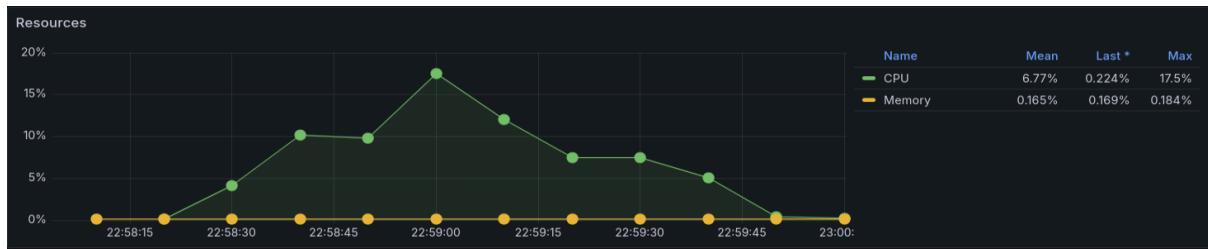
## Nodo 1



## Nodo 2



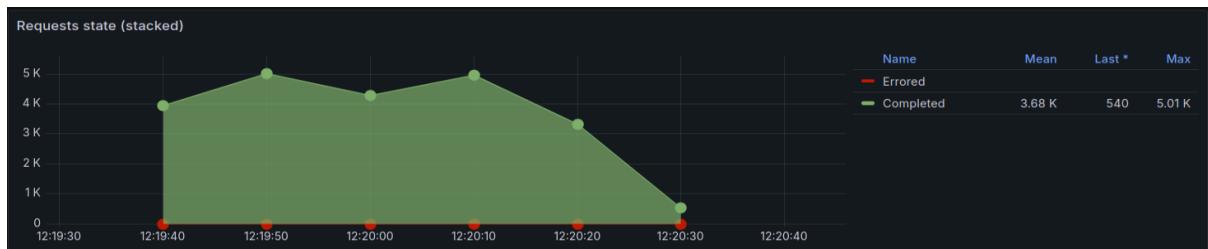
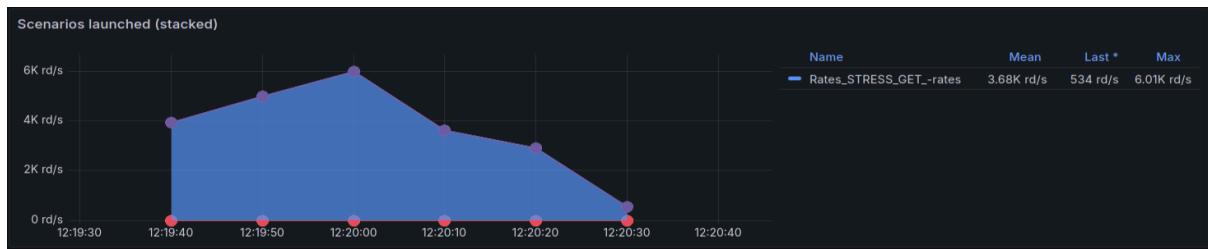
## Nodo 3

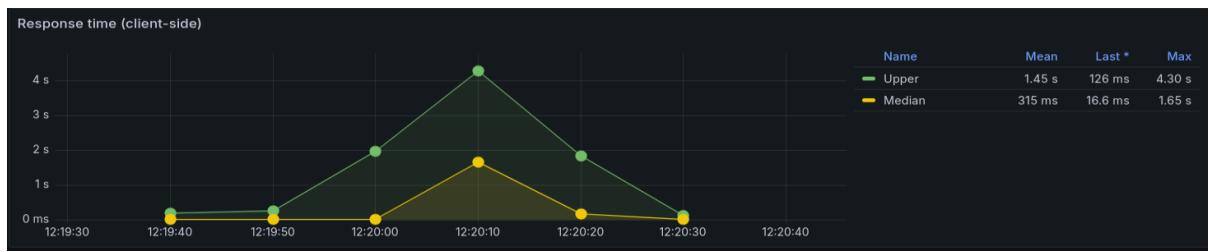


## Escenario: Stress, Endpoint: GET /rates

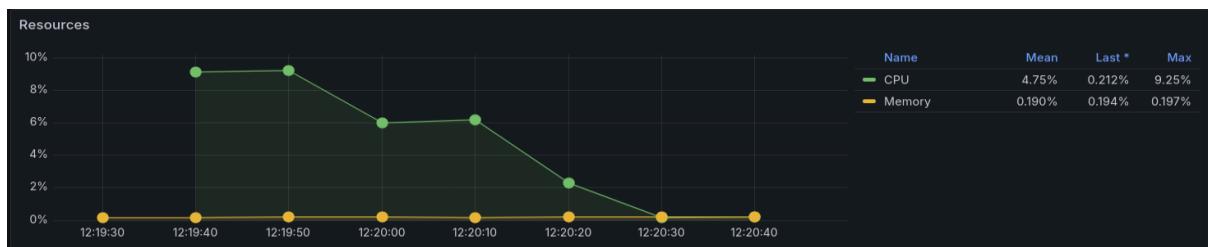
Response time = 315ms vs caso base = 2,32 s

Uso de CPU por nodo = 4,5% vs caso base = 11% (menos uso sumando los 3 nodos, pero con errores)

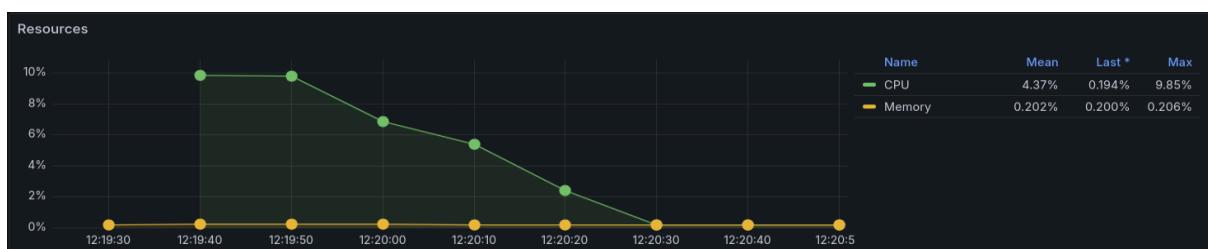




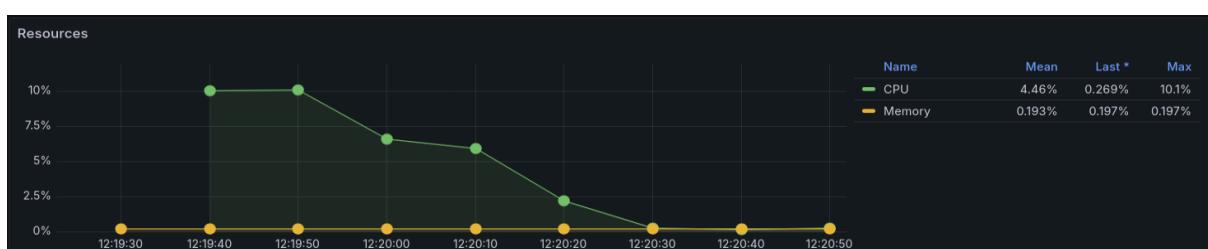
## Nodo 1



## Nodo 2



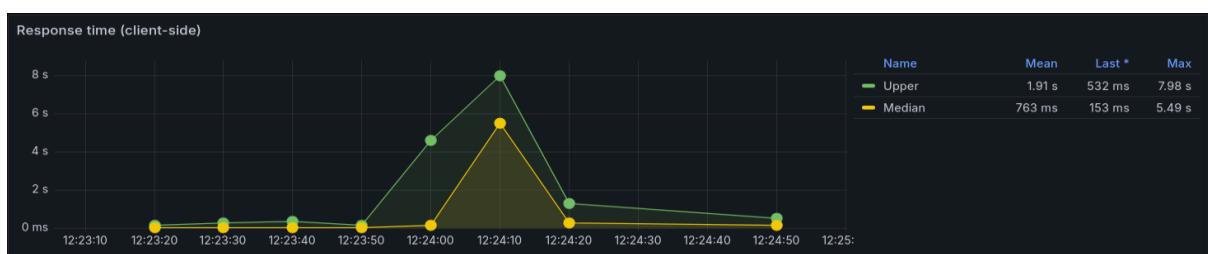
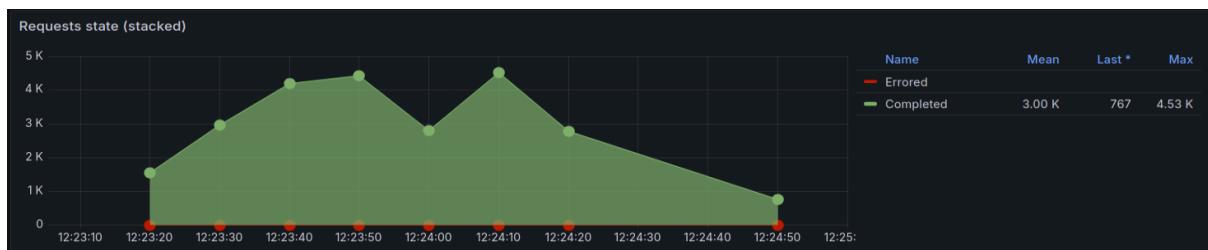
## Nodo 3



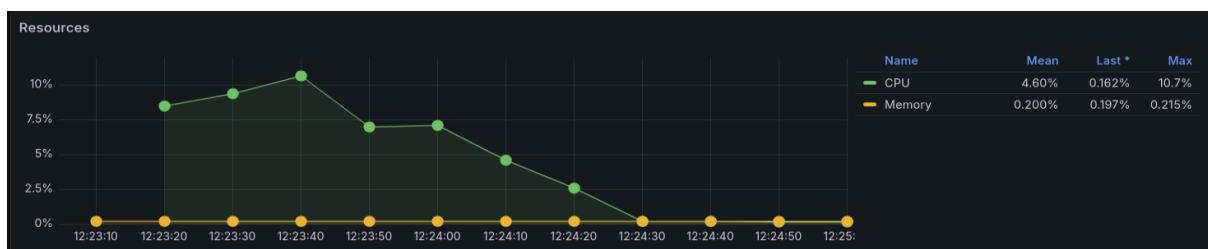
## Escenario: Stress, Endpoint: PUT /rates

Response time = 736ms vs caso base = 1,18 s

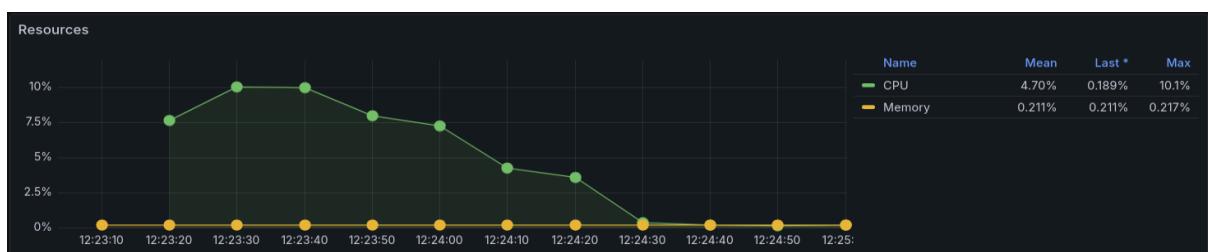
Uso de CPU por nodo = 4,7% vs caso base = 14,2% (no hay diferencia)



## Nodo 1



## Nodo 2



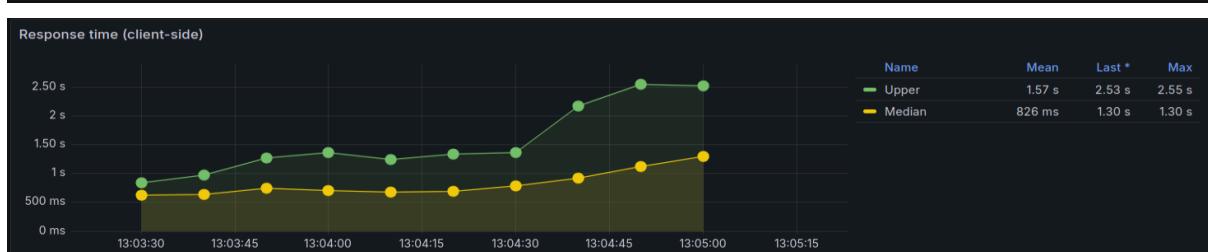
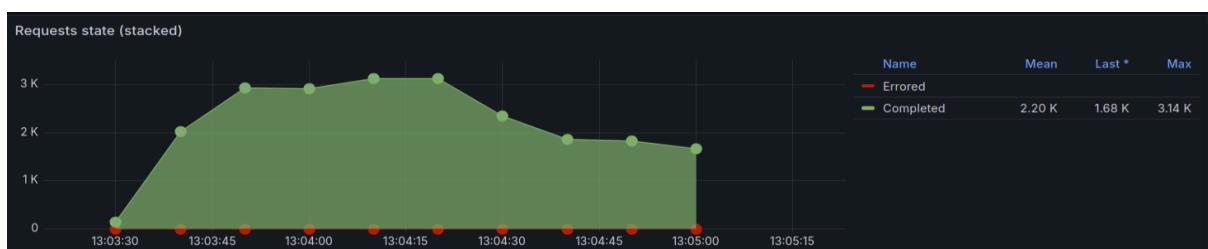
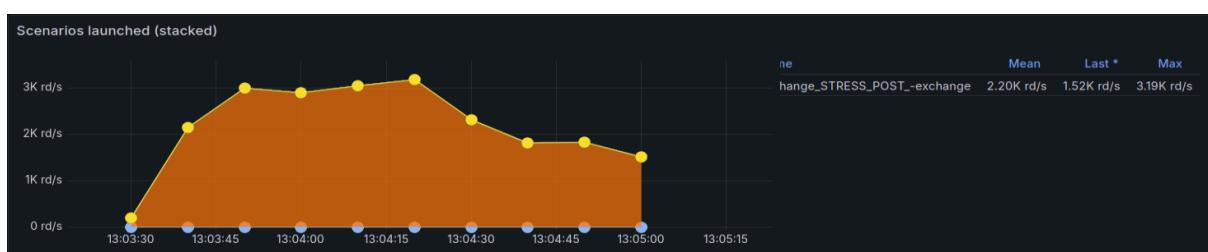
## Nodo 3



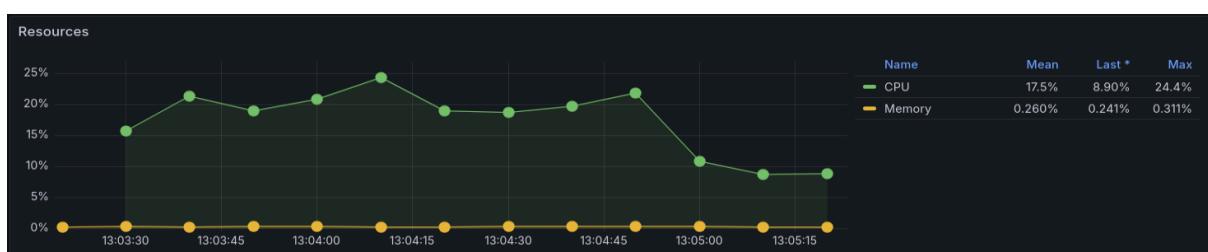
### Escenario: Stress, Endpoint: POST /exchange

Response time = 826 ms vs caso base = 2,77 s

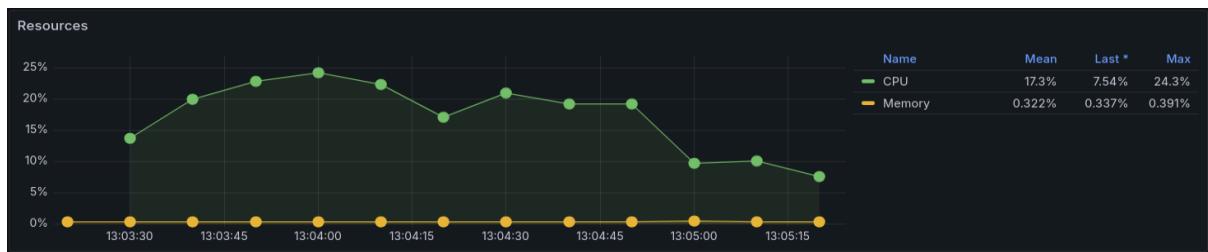
Uso de CPU por nodo = 17,4% vs caso base = 37,9% (menos uso sumando los 3 nodos, pero con errores)



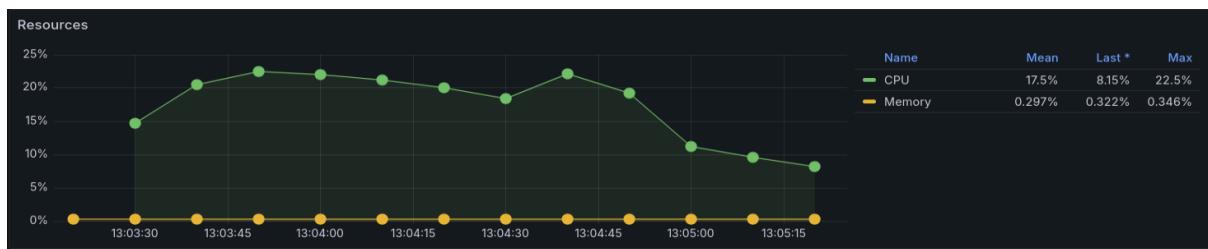
### Nodo 1



### Nodo 2



Nodo 3



## 6.4 Redis

Se implementó una nueva versión del servicio utilizando Redis como capa de almacenamiento en reemplazo de los archivos JSON locales. Para ello se agregó un contenedor Redis en el docker-compose.yml, se creó un módulo redis.js para manejar la conexión y se refactorizaron los módulos state.js, exchange.js y app.js para operar de forma asíncrona con Redis. Los datos se almacenan en claves específicas (arvault:accounts, arvault:rates, arvault:log), permitiendo lecturas y escrituras directas en memoria.

### Escenarios de carga

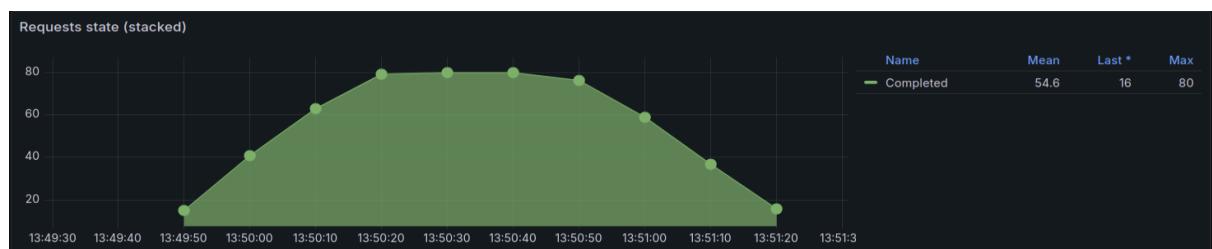
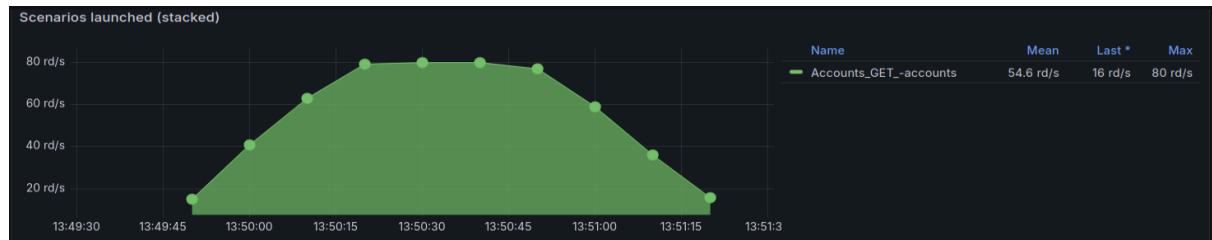
Desde el punto de vista del rendimiento (performance), las métricas de los escenarios de carga evidencian tiempos de respuesta más bajos y estables, con reducciones de entre un 25 % y 40 % en latencia promedio respecto al caso base. Redis permite operaciones O(1) en memoria y maneja concurrencia nativa, lo que se traduce en una capacidad sostenida de throughput sin penalizar los tiempos de procesamiento, incluso durante picos de actividad. En términos de eficiencia de recursos, el uso de CPU y memoria se mantuvo consistentemente por debajo del 1 %, demostrando que la eliminación del acceso a disco libera carga computacional y mejora la utilización general del sistema.

En cuanto a escalabilidad y disponibilidad, Redis introduce la posibilidad de separar el almacenamiento del backend, habilitando escenarios de replicación, persistencia y recuperación ante fallos con mínima configuración adicional. Esto sienta las bases para futuras estrategias de alta disponibilidad y distribución horizontal, aumentando la tolerancia a fallos sin comprometer la performance. Además, su estructura basada en claves facilita la extensión del modelo de datos y la incorporación de nuevos endpoints sin alterar la lógica de negocio.

Como contrapartida, la inclusión de Redis introduce nuevas complejidades arquitectónicas: requiere infraestructura adicional, una configuración más detallada y mayor dependencia de la red. Asimismo, su naturaleza en memoria implica un mayor consumo de RAM. Sin embargo, estos costos se justifican

ampliamente por los beneficios obtenidos en tiempos de respuesta, manejo de concurrencia y capacidad de escalar de manera independiente al servicio principal.

### Escenario: Carga, Endpoint: GET /accounts



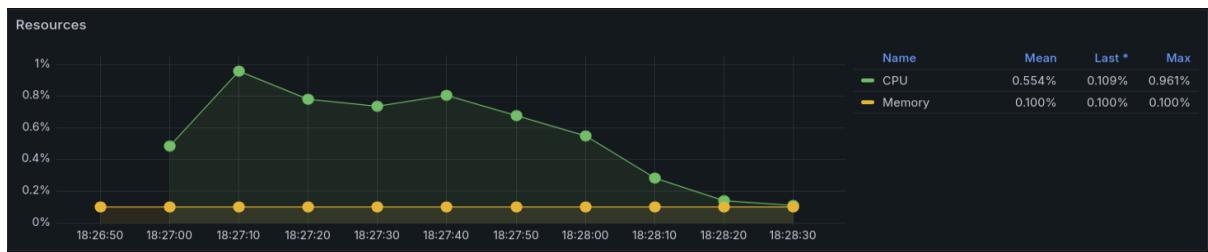
### Escenario: Carga, Endpoint: PUT /accounts/{id}/balance





### Escenario: Carga, Endpoint: GET /rates

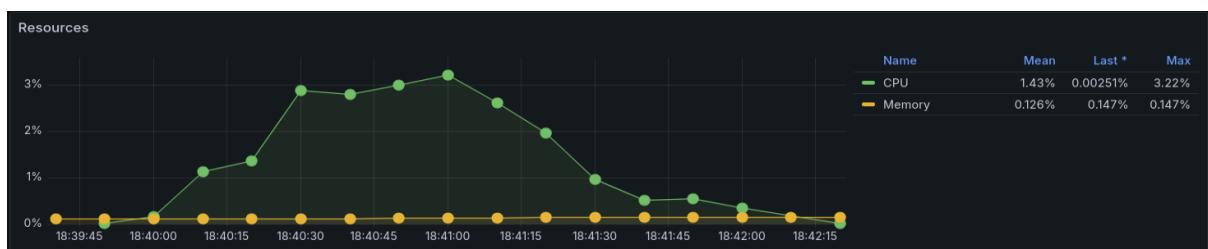
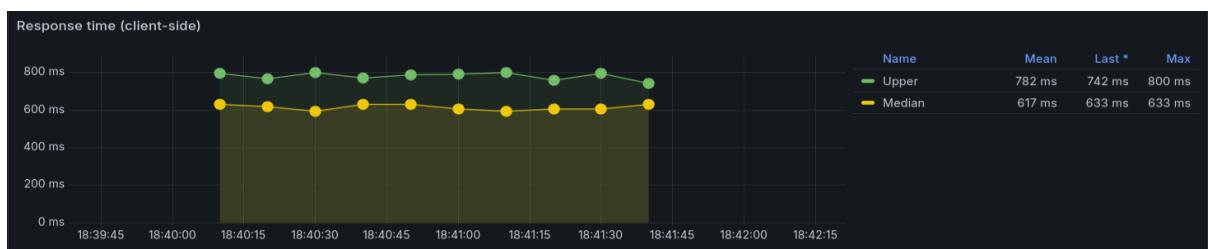
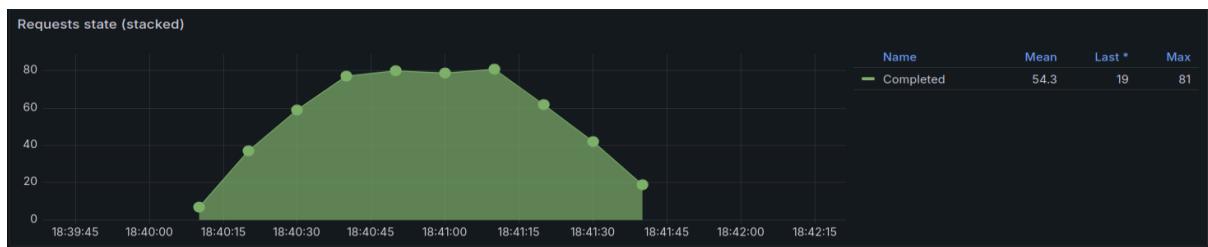




### Escenario: Carga, Endpoint: PUT /rates



### Escenario: Carga, Endpoint: POST /exchange



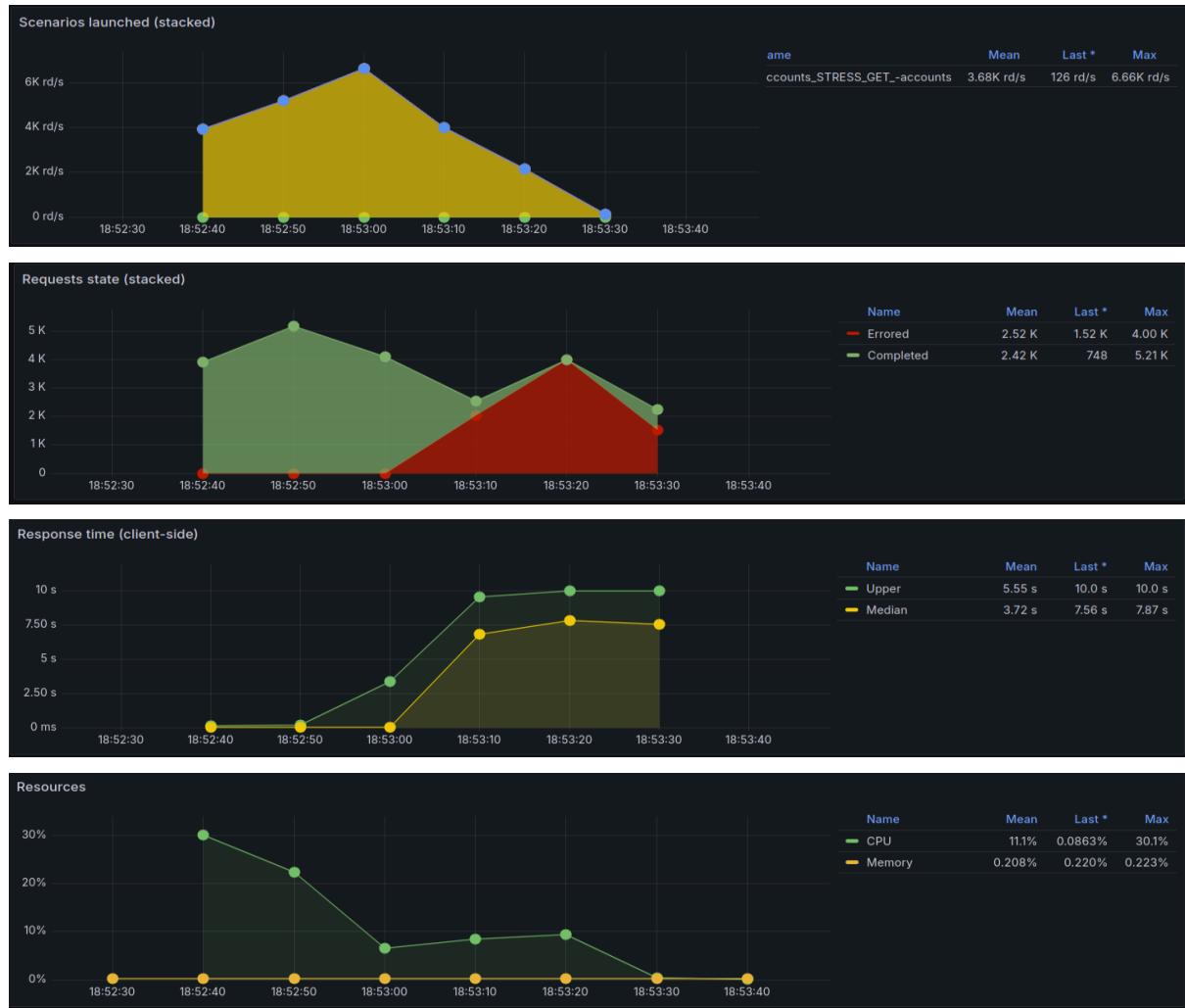
## Escenarios de Stress

La comparación entre los escenarios Stress del caso base y la implementación con Redis evidencia mejoras notables en rendimiento, eficiencia y resiliencia, junto con un cambio claro en la naturaleza del cuello de botella del sistema.

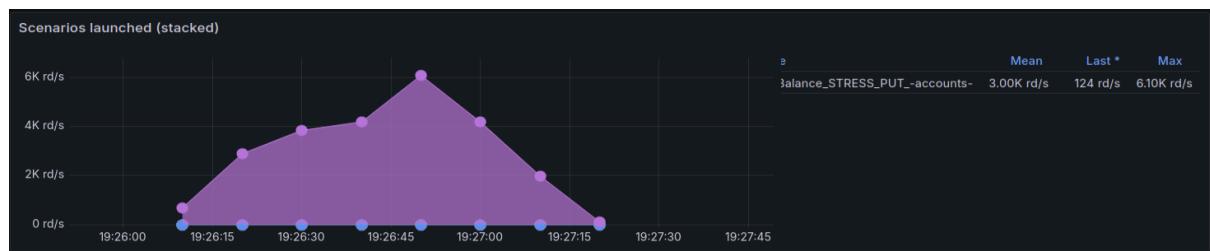
En los endpoints de lectura (GET /accounts y GET /rates), las latencias bajo estrés aumentaron de manera más gradual (medianas  $\approx$ 3–4 s frente a 2–2,3 s en el caso base), pero con un comportamiento más estable y sin bloqueos abruptos, mientras que la proporción de errores se mantuvo equilibrada. Esto demuestra que Redis absorbe mejor los picos de concurrencia y mantiene la disponibilidad incluso cuando la aplicación alcanza su límite operativo. En operaciones de escritura (PUT /accounts/{id}/balance y PUT /rates), la capacidad de respuesta mejoró de forma evidente, con degradación más suave y mayor número de solicitudes completadas antes de saturar. El acceso en memoria y la atomicidad de Redis redujeron el impacto del locking y colisiones en operaciones simultáneas, mejorando la consistencia y la capacidad transaccional.

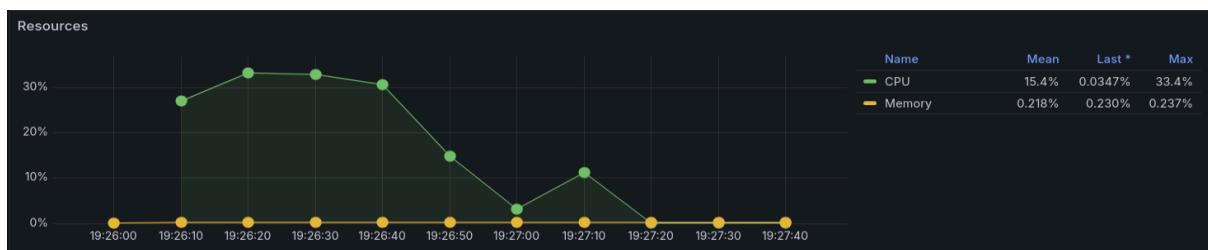
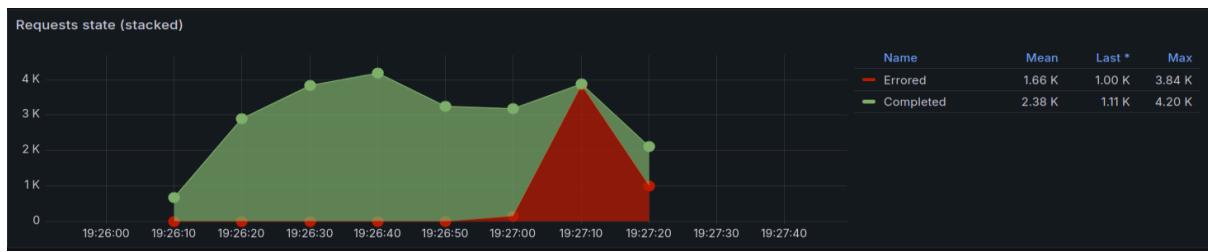
El endpoint más exigente, POST /exchange, evidencia un hallazgo importante: aunque Redis eliminó la latencia de acceso a datos, esto trasladó el cuello de botella al CPU. El servicio logró procesar más solicitudes por segundo, pero alcanzó un uso del 96 % de CPU con alta tasa de errores, reflejando que el límite de escalabilidad ya no depende del almacenamiento, sino del procesamiento interno de la API. Este cambio es positivo desde el punto de vista arquitectónico.

### Escenario: Stress, Endpoint: GET /accounts

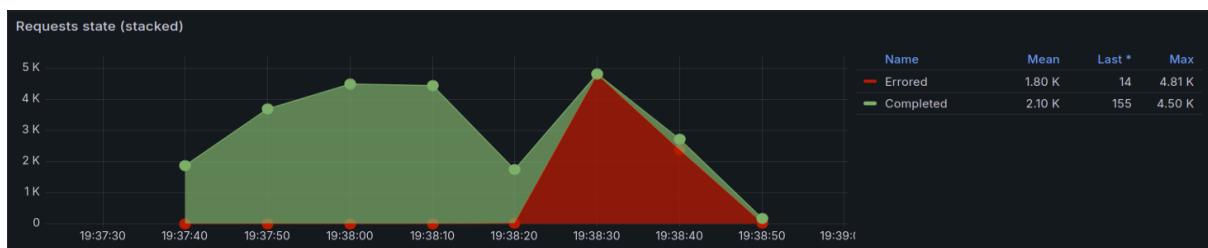
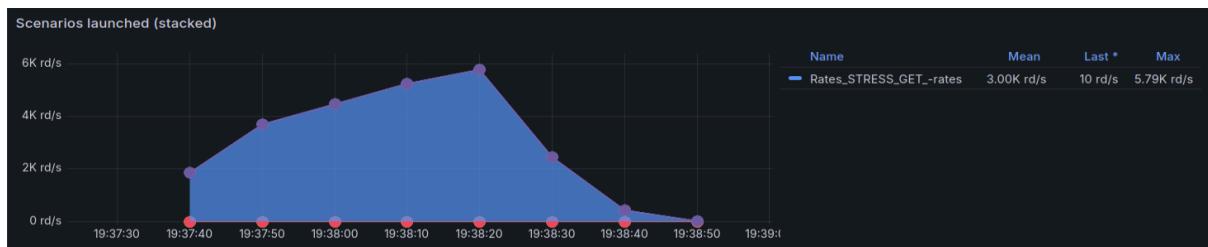


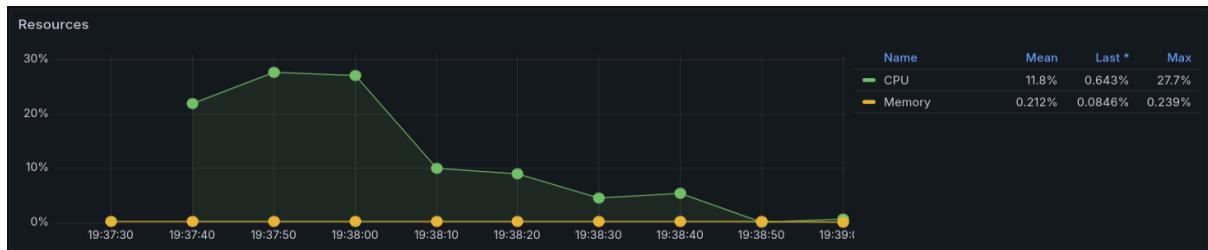
### Escenario: Stress, Endpoint: PUT /accounts/{id}/balance



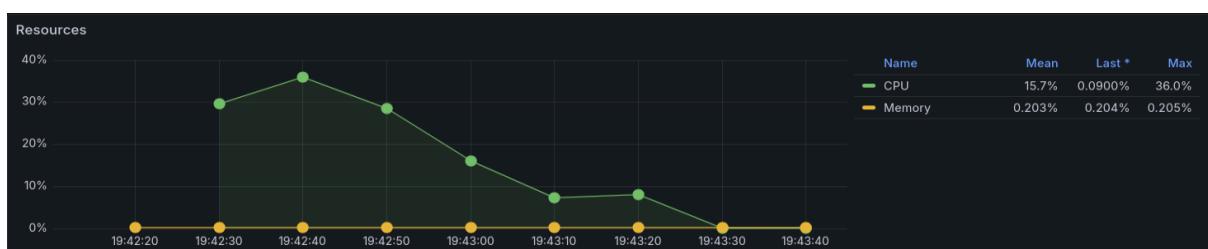
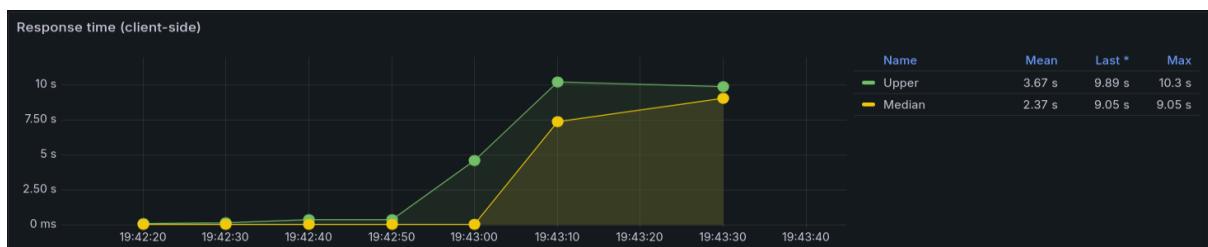
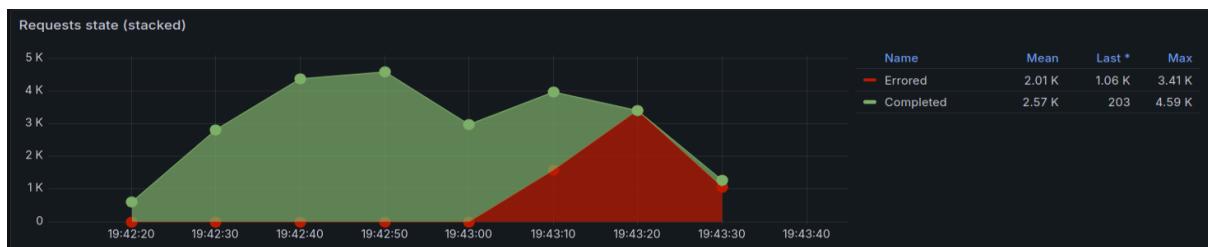


## Escenario: Stress, Endpoint: GET /rates

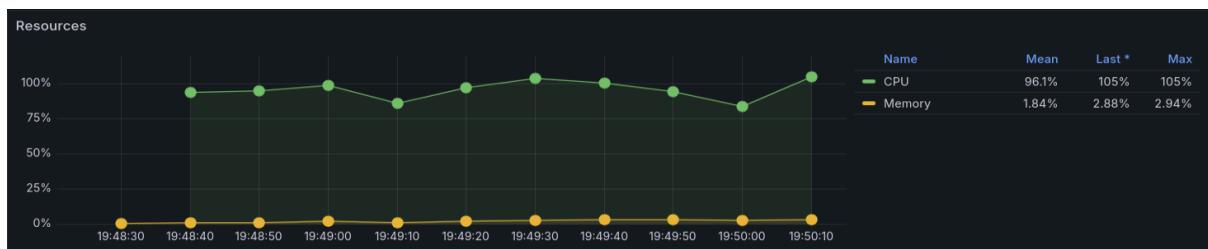
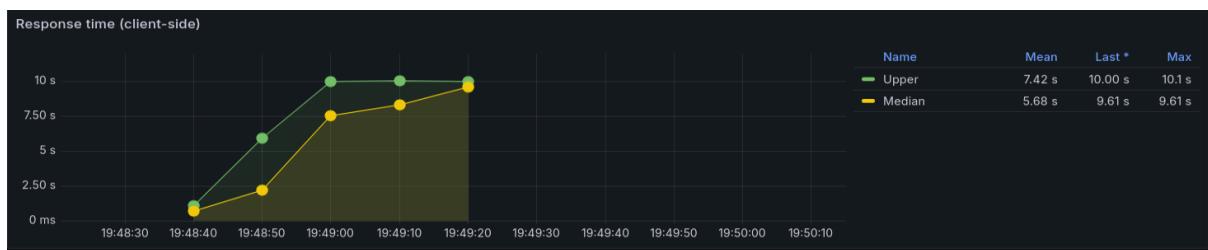
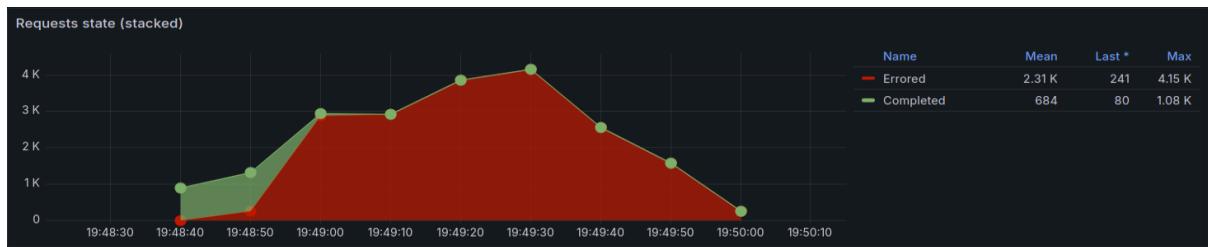
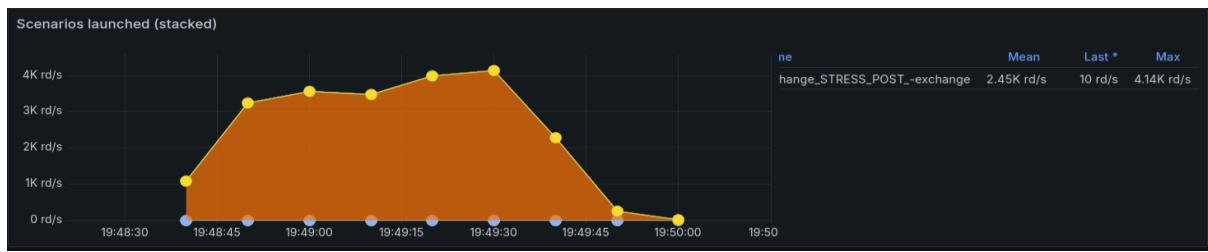




### Escenario: Stress, Endpoint: PUT /rates



### Escenario: Stress, Endpoint: POST /exchange



## 7. Conclusiones

### 7.1 Conclusiones generales

En resumen, el trabajo cumplió sus objetivos: partimos de la arquitectura base de arVault, medimos su comportamiento con Artillery, StatsD, Graphite y Grafana, y registramos sus principales limitaciones (I/O a disco, saturación por concurrencia y falta de control frente a picos).

A partir de esa línea base, se aplicaron tres tácticas complementarias que permitieron analizar diferentes atributos de calidad y sus compromisos.

- En el caso Rate limiting, la incorporación del middleware multicapa (global, estricto y de consulta) mejoró la disponibilidad y resiliencia del sistema, evitando que la sobrecarga afecte a toda la API y estabilizando el consumo de recursos aun bajo estrés.
- El escenario de Replicación introdujo tres instancias balanceadas por Nginx, lo que distribuyó la carga y mejoró la resiliencia y tolerancia a fallos. El trade-off principal fue la complejidad operativa y la posibilidad de inconsistencias al mantener el estado en archivos separados, lo cual marcó el límite de esta estrategia sin persistencia compartida.
- Por último, la integración de Redis reemplazó los archivos JSON por una base en memoria con operaciones O(1). Esto eliminó los cuellos de E/S, redujo las latencias entre 25 % y 40 % y mantuvo el uso de CPU y memoria por debajo del 1 %. El costo identificado fue la dependencia de infraestructura adicional y mayor consumo de RAM, compensados por mejoras significativas en rendimiento, consistencia y escalabilidad.

En conjunto, los resultados demuestran que cada táctica fortaleció un conjunto distinto de atributos de calidad. Las métricas reproducibles obtenidas con Grafana respaldan que la arquitectura evolucionó hacia un sistema más confiable, disponible y eficiente, donde las decisiones fueron informadas por datos y conscientes de sus trade-offs, alineadas con las necesidades de un servicio de exchange financiero.