Informe de Arquitectura arVault

Integrantes

Nombre	Padrón	Correo electrónico
Alejandro Nicolás Smith	101730	asmith@fi.uba.ar
Guido Menendez	109279	gmenendez@fi.uba.ar
Hernán Mario Lagarde	105827	hlagarde@fi.uba.ar
Patricio Galvan	106166	pggalvan@fi.uba.ar

Repositorio GitHub: https://github.com/aleesmitth/arq-software-1c25-tp-1/tree/main

---

Introducción

Este informe tiene como objetivo auditar la arquitectura actual del sistema, identificar cuellos de botella que impactan en atributos de calidad clave y proponer mejoras fundamentadas mediante tácticas arquitectónicas y pruebas empíricas. Para ello, se analizará el código fuente, se realizarán pruebas de carga con herramientas como Artillery y Docker, y se implementarán modificaciones estratégicas para validar su impacto en métricas críticas como tiempo de respuesta, consumo de recursos y consistencia de datos.
El objetivo final es proveer una hoja de ruta técnica que permita a arVault recuperar la confianza de usuarios e inversores.

Disclaimers

- Para evitar ser redundantes con el contenido del readme, no se especifican detalles de los endpoints expuestos por el servicio de billetera virtual.

Atributos de Calidad Principales

1. Availability (Disponibilidad)

Separamos la disponibilidad en función de las dos funcionalidades principales:

- *Billetera virtual*: debe tener la disponibilidad más alta posible. Los usuarios deben poder consultar y disponer de sus fondos en cualquier momento. Una caída de este servicio impacta directamente en la confianza del usuario.
- *Intercambio de divisas*: si bien es deseable que esta funcionalidad esté siempre disponible, su caída no afecta al usuario de la misma manera crítica que la billetera virtual. Sin embargo, puede generar pérdida de oportunidad económica.

2. Scalability (Escalabilidad)

Es fundamental dado el perfil B2C del producto. Un crecimiento en la cantidad de usuarios debe poder ser acompañado sin degradar el servicio. Actualmente la arquitectura limita totalmente esta capacidad.

3. Observability (Observabilidad / Visibilidad)

Debe ser posible:

- monitorear comportamiento del sistema,
- entender fallas,
- trazar transacciones específicas (para resolver reclamos o cuestiones legales).

La auditoría del sistema es vital, especialmente porque el enunciado menciona reclamos frecuentes de usuarios.

4. Security (Seguridad)

Relevante tanto para proteger datos sensibles (fondos, movimientos) como para proteger el sistema de abusos. Algunos puntos concretos:

- Rate limiting.
- Persistencia segura de datos sensibles.
- No repudio de transacciones.

5. Interoperability (Interoperabilidad)

Capacidad del sistema de integrarse con otros servicios externos:

- Consultar cotizaciones de divisas.
- Realizar transferencias.
- Integrarse con bancos o sistemas de pago.

Atributos de Calidad Secundarios

1. Performance

En este tipo de sistema, la performance debe ser razonable pero no es crítica salvo en operaciones específicas (ej: cotización de divisas en tiempo real). La mayor preocupación está en la escalabilidad y disponibilidad.

2. Maintainability (Mantenibilidad)

Facilidad para mantener y extender el sistema en el tiempo.

Análisis del Sistema

Componentes:

Componente	Rol	Ubicación Operativa
Nginx	Reverse Proxy + Load Balancer	Docker container propio
Servicio (Node.js)	Servicio principal con funcionalidades de billetera virtual e intercambio de divisas	Docker container propio
Archivos JSON	Persistencia de datos	FileSystem del servicio principal

Diagrama de Arquitectura Funcional

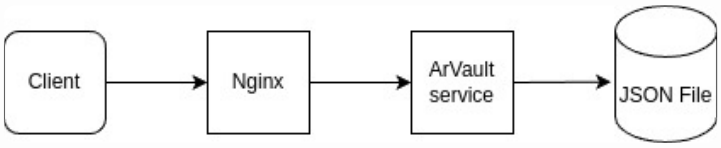
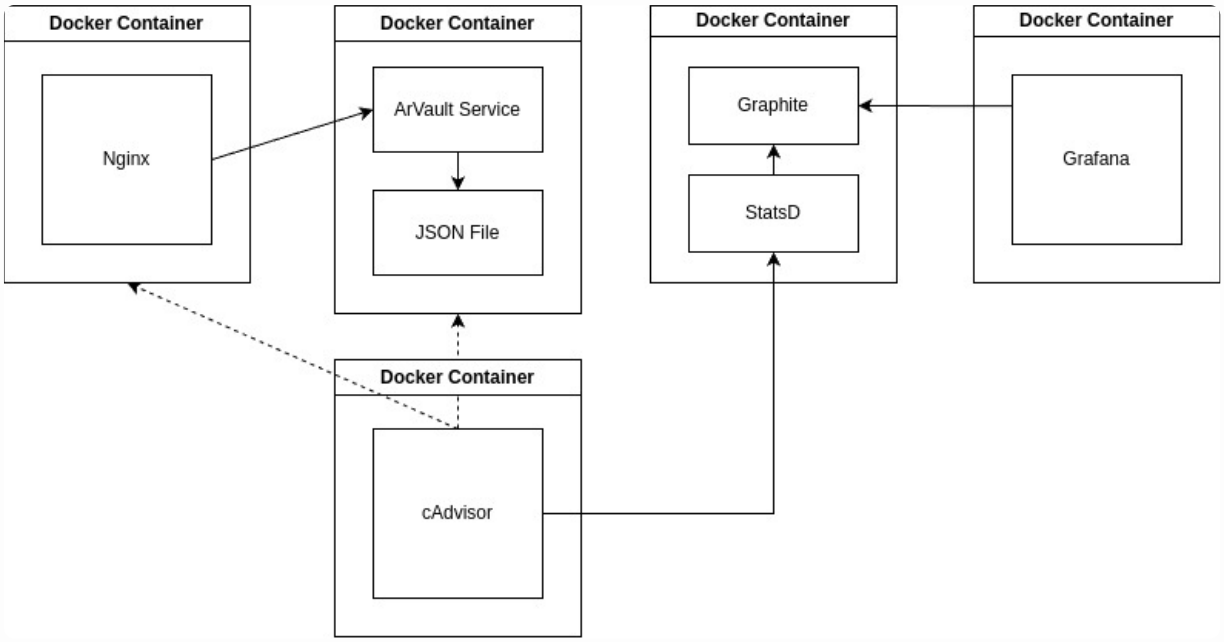


Diagrama de Arquitectura Operativa



Observaciones

Observación	Impacto	Solución Propuesta
SPOF en NGINX	Caída de NGINX deja fuera de servicio todo.	Redundancia / Utilizar un load balancer o reverse proxy de alguna plataforma de nube que garantice alta disponibilidad.
Falta de red privada	Cualquier contenedor es accesible directo por IP interna.	Usar red interna de Docker. Sólo NGINX expuesto.
Procesamiento de requests del servicio mediante modelo Async de Node.js	Ideal para aplicaciones con muchas operaciones I/O, baja performance en aplicaciones con muchas operaciones de alto cómputo.	No parece ser un problema ya que no se identifican operaciones de cómputo pesado dentro de la lógica del servicio, a excepción de aquellas que serán mencionadas más adelante como "*operaciones de datos*" que no deberían ser responsabilidad del servicio, por lo que no las consideramos como parte de la lógica de negocio.
El servicio principal es un monolito	No permite aplicar estrategias de arquitectura diferentes	Separar las funcionalidades en distintos servicios.

	según las necesidades de cada funcionalidad. Es imposible escalar una funcionalidad por separado, o aplicar un modelo de concurrencia diferente para cada funcionalidad, o priorizar disponibilidad según funcionalidad. Convierte al servicio principal en un SPOF.	
Persistencia en archivos JSON	- **No atomicidad:** Las escrituras y lecturas no son atómicas porque se utiliza un modelo de estado mutuo compartido sin mecanismos de exclusión mutua como locks, lo que expone el sistema a condiciones de carrera cuando múltiples procesos acceden concurrentemente al mismo archivo. Si hay múltiples requests modificando los mismos datos, habrá corrupción (ej: 2 operaciones de escritura simultáneas en accounts.json). - **Escalabilidad:** Imposible escalar horizontalmente (otras instancias de Node.js no compartirán el estado). - **Durabilidad:** Si el servidor crashea antes de un scheduleSave, se pierden datos.	Localizar base de datos en un servicio externo utilizando un sistema de gestión de base de datos.
El módulo exchange.js	**- Consumo Excesivo de Memoria (RAM)**: En escenarios	Utilizar una caché en memoria como Redis para

almacena todas las transacciones en un array en memoria, que luego se persiste periódicamente en el archivo log.json.	de alta carga, se generan muchos logs que serán almacenados de forma temporal en RAM. **- Persistencia Frágil:** El array se guarda en log.json cada 1 segundo mediante *scheduleSave*. Si el servidor se cae antes del próximo guardado, se pierden las transacciones no guardadas.	almacenar logs temporalmente, con una política de traspaso periódico a una base de datos persistente en disco. Alternativamente, emplear un servicio externo especializado en gestión de logs.
El módulo state.js carga todos los archivos JSON en memoria RAM al iniciar el servidor	Si se levanta un archivo de gran tamaño, esto puede llevar a delays hasta tener el sistema listo para operar, además de consumos excesivos de RAM.	No utilizar persistencia en archivos.
Operaciones de datos se realizan en memoria sobre el servicio principal	Bloqueo del hilo principal de Node.js, imposibilidad de usar índices.	Delegar las operaciones de datos como filtrados a un sistema de gestión de base de datos
Datos sensibles persistidos en texto plano.	Vulnerabilidades de seguridad.	Encriptar datos sensibles antes de persistirlos.
Exchange rates hardcodeados	No reflejan cotización actualizada a tiempo real.	Integrarse con un servicio externo para obtener datos en tiempo real.
Exchange rates persistidos de forma redundante	La redundancia hace que se almacene más información de lo necesario.	Tomar una moneda como base y guardar las demás monedas relativas a la base.

Importante: *Antes de tratar cualquiera de estas observaciones como un problema a resolver, se recomienda evaluar si realmente representan una limitación significativa dentro del contexto operativo de la aplicación y si justifican la inversión en su solución.*

Escenarios de prueba sobre arquitectura base

Para evaluar el comportamiento de la arquitectura tal como fue recibida, se definieron **3 escenarios distintos de carga para cada uno de los 4 endpoints** disponibles en el servicio. Los escenarios fueron configurados en archivos `.yaml` utilizando Artillery y enviando métricas a Graphite mediante el plugin `statsd` y cAdvisor, visualizadas luego en Grafana.

Los escenarios son:

- `base`: carga baja pero sostenida. Útil como línea base para comparar contra versiones optimizadas del sistema.
- `load`: carga media con múltiples requests por usuario simulados. Representa un uso concurrente sostenido y realista.
- `peak`: pico breve de tráfico intenso. Permite evaluar la capacidad del sistema ante eventos súbitos de alta demanda.

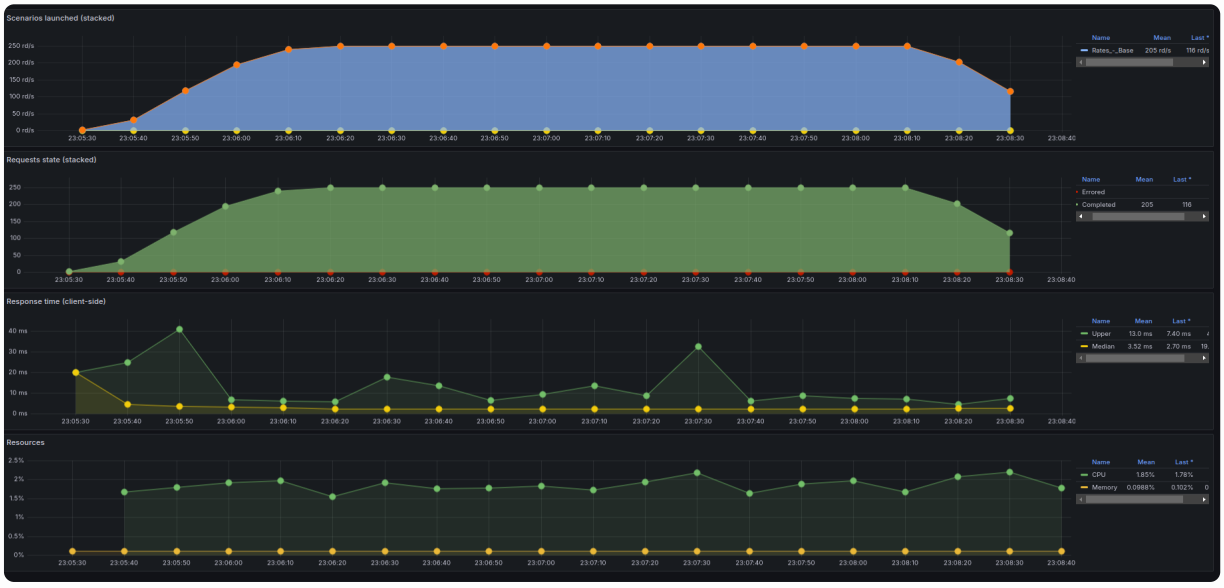
> Se definieron tiempos prolongados (entre 3 y 4 minutos por prueba) para obtener mayor resolución en los gráficos y facilitar un análisis más preciso del comportamiento a lo largo del tiempo.

Resultados obtenidos

A continuación se presentan **ejemplos representativos**, cubriendo al menos una vez cada endpoint y cada tipo de escenario:

Endpoint: `/rates` – Escenario `base`

Carga controlada con ramp-up y estabilidad. Permite observar el comportamiento en condiciones ideales.

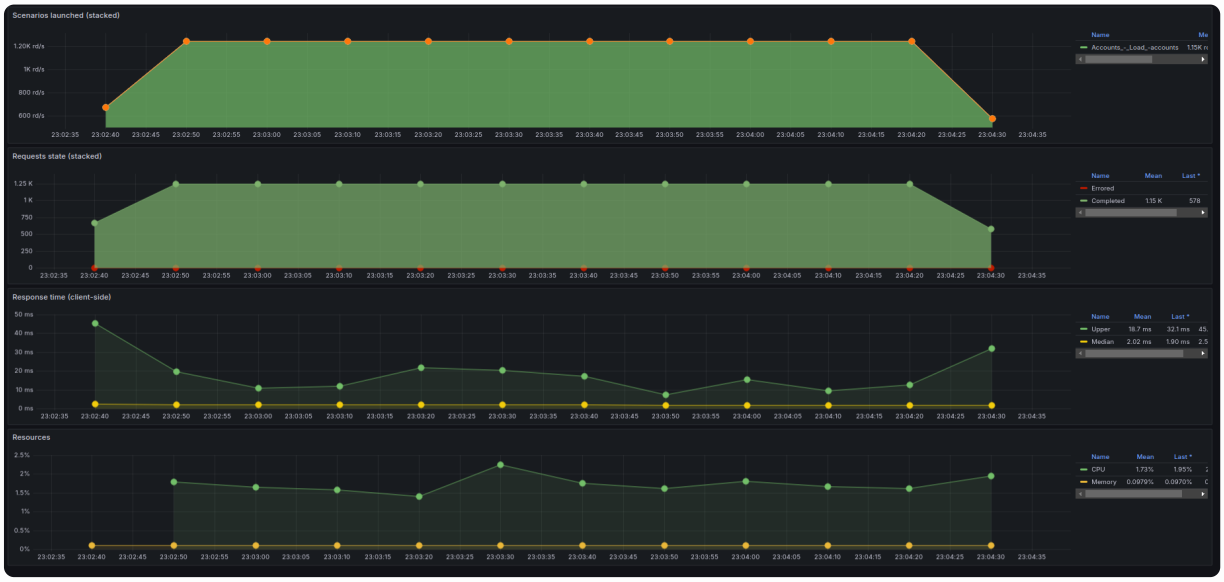


Observaciones:

- Comportamiento estable y sin errores.
- Latencias por debajo de 5ms durante toda la prueba.
- Bajo consumo de CPU (<1%) y memoria (<0.1%).

Endpoint: `/accounts` – Escenario `load`

Se incrementa la carga simulando un uso más realista. Cada usuario realiza múltiples requests.

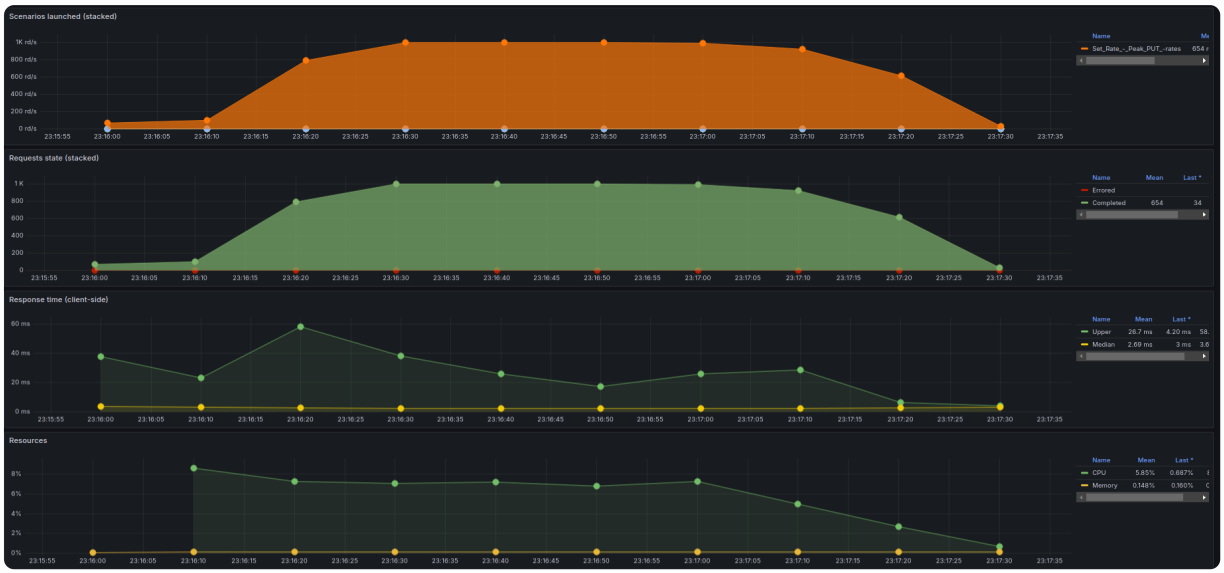


Observaciones:

- Buen escalamiento hasta ~1.2k rps.
- Respuesta constante con muy baja latencia media (~2ms).
- CPU apenas supera el 2%, lo que indica buen margen de escalabilidad.

Endpoint: `PUT /rates` – Escenario `peak`

Pico de carga breve e intenso para observar el sistema en condiciones extremas.

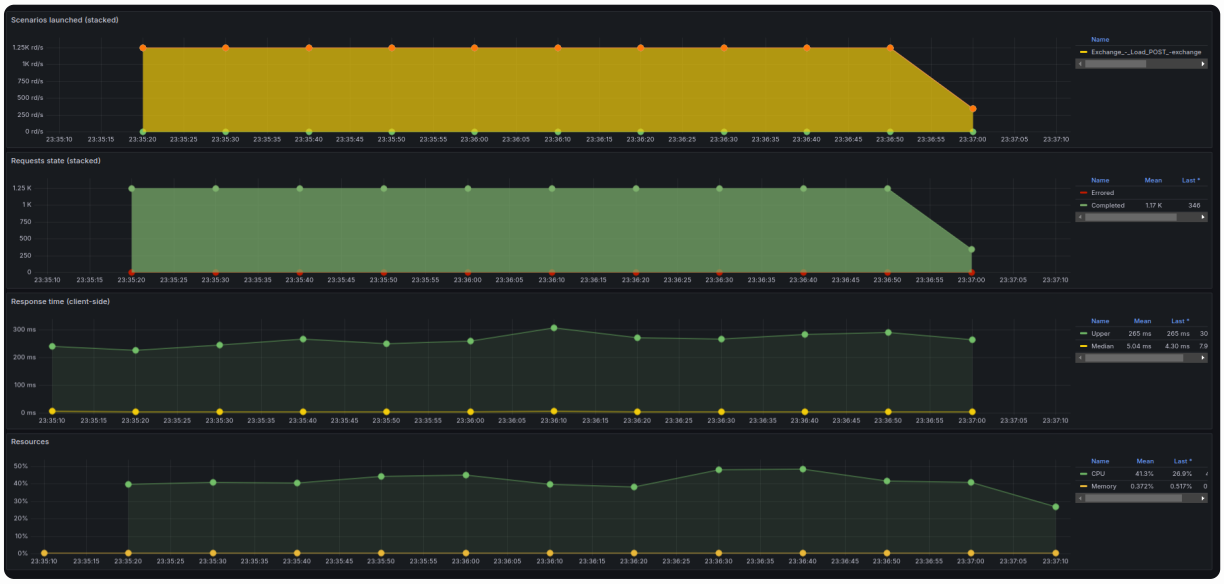


Observaciones:

- Carga cercana a los 1k rps durante el pico.
- Latencia inicial elevada que luego se estabiliza rápidamente.
- CPU y memoria en rangos normales, sin errores registrados.

Endpoint: `POST /exchange` **– Escenario** `load`

Endpoint más computacionalmente complejo, involucra múltiples validaciones y escritura de datos.



Observaciones:

- Latencia notablemente superior (hasta ~300ms), como era esperable.
- CPU alcanza picos de 40% debido al procesamiento adicional.
- A pesar de la carga, no se registraron errores ni saturaciones.

Conclusión de los resultados obtenidos

Los resultados muestran que:

- El sistema responde correctamente en todos los endpoints bajo condiciones de carga típicas (`base`) y sostenidas (`load`).
- Se identifica claramente al endpoint `/exchange` como el más costoso computacionalmente.
- Los gráficos de `peak` muestran la recuperación del sistema una vez superado el pico, sin errores, lo cual es positivo.
- No se detectaron cuellos de botella críticos, aunque algunos endpoints requieren especial atención a medida que se escale.

Estas pruebas sientan una base objetiva sobre la cual se aplicarán tácticas de mejora (replicación, rate limiting, etc.), evaluando su impacto en estos mismos escenarios.

Estrategias

A continuación se listan y explican en detalle las estrategias implementadas para mejorar el sistema.

1. Persistencia en RedisCloud

Se cambió la unidad de persistencia de archivos JSON a RedisCloud. Esto soluciona los siguientes problemas (detallados ya en la sección de observaciones):

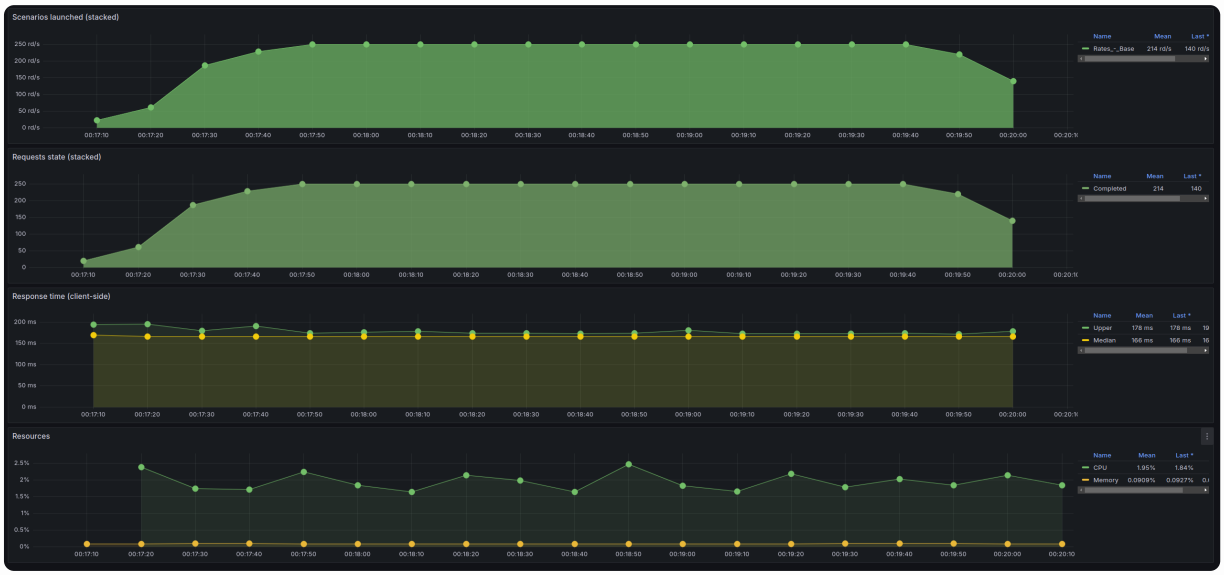
1. Imposibilidad de escalamiento horizontal.
2. Concurrencia insegura.
3. Durabilidad pobre.
4. Necesidad de persistir datos de forma periódica.
5. Inicio lento del servidor debido a la necesidad de levantar todos los archivos a memoria.

Es importante aclarar que:

1. El sistema sigue realizando las tareas de filtrado y búsqueda de datos sobre las colecciones completas en el servicio principal, por lo que los problemas relacionados con esto siguen presentes.

Analisis de graficos Grafana

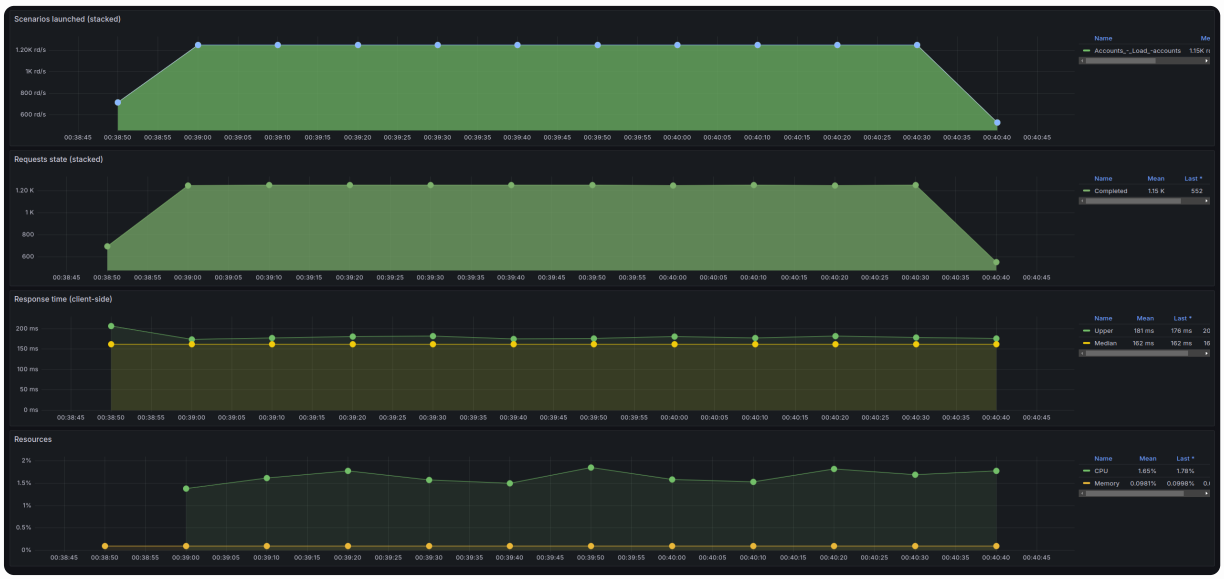
Endpoint: GET /rates
Escenario: Base



- **Consistente:** los valores son iguales en mean y last, lo que indica estabilidad en los tiempos.
- **Poca variabilidad:** como el valor "Upper" está muy cerca del de la mediana, no hay mucha dispersión — es decir, todos los requests están tardando bastante similar y de forma relativamente alta.

Endpoint: GET /accounts

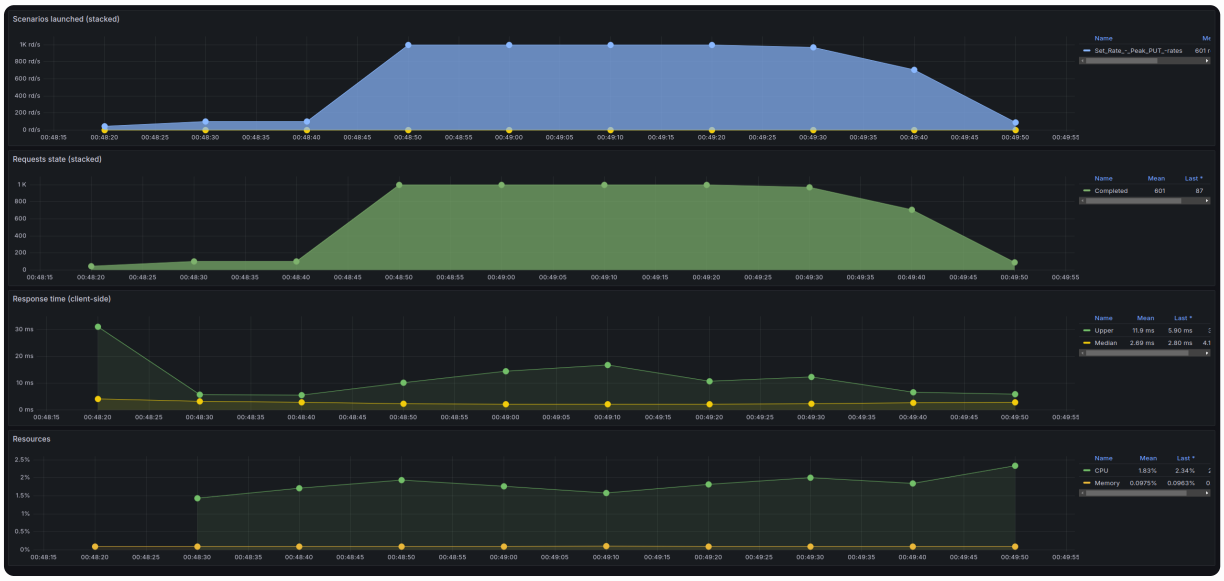
Escenario: Load



- Similar al endpoint anterior, pero con un "last Upper" más bajo que el promedio (176ms vs 181ms), lo que podría indicar una leve mejora reciente.
- En general, los tiempos siguen siendo altos, aunque consistentes.

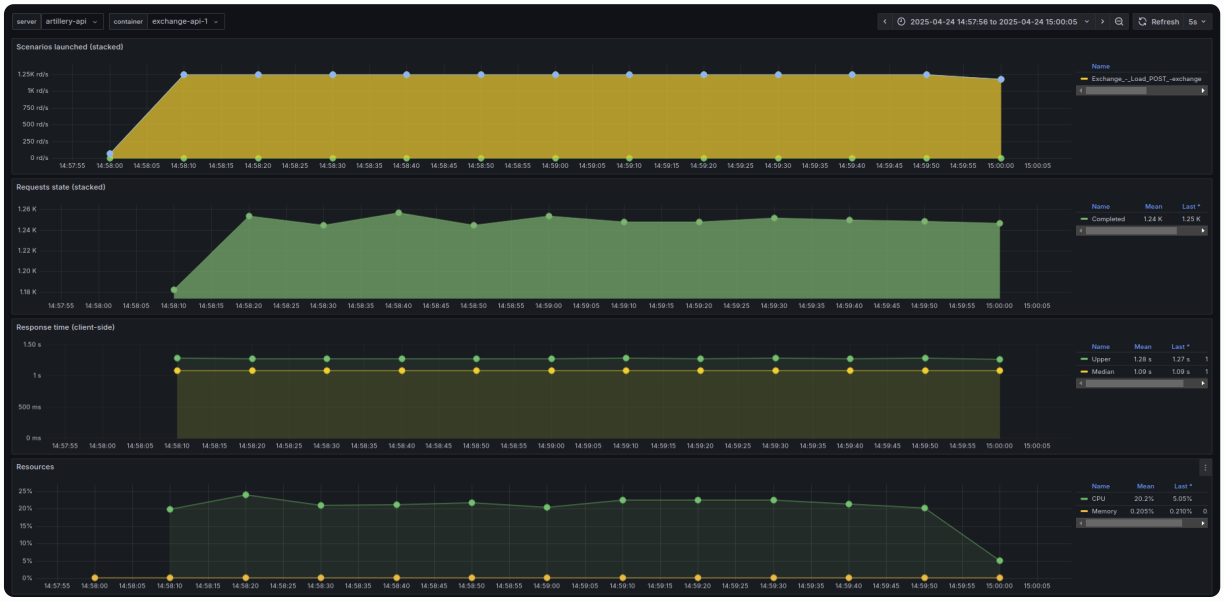
Endpoint: PUT /rates

Escenario: Peak



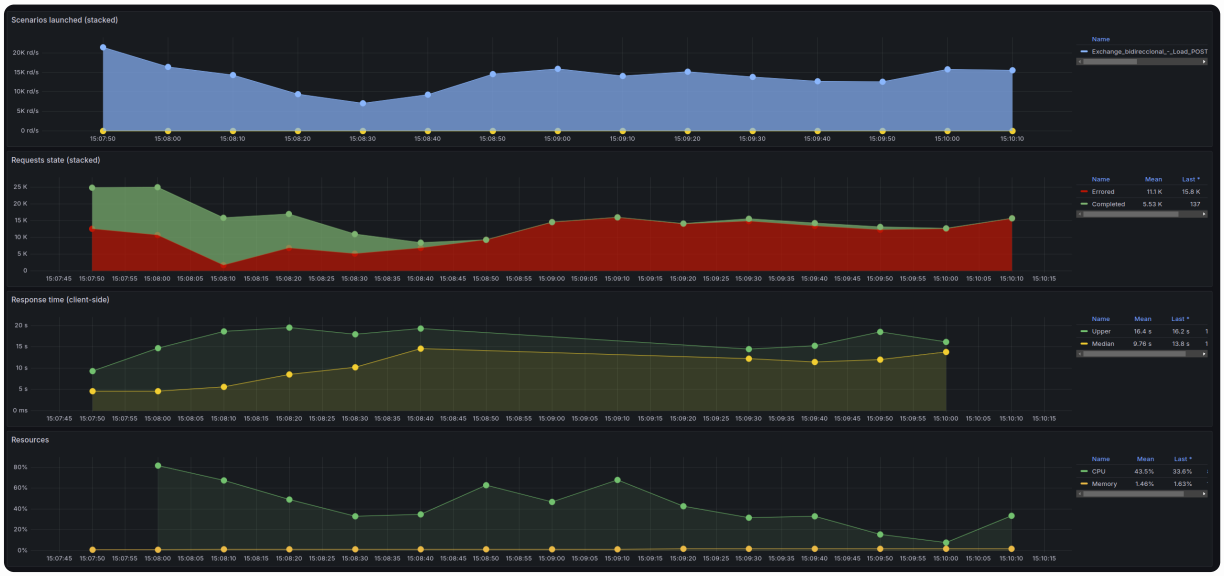
- Excelente performance: tanto la mediana como el upper son bajos.
- Últimos valores aún mejores: el último "Upper" bajó a 5.9ms, indicando mejora o menor carga reciente.

Endpoint: POST /exchange
Escenario: Load



- Mediana alta: 1.09 segundos significa que al menos la mitad de las respuestas están tardando más de un segundo. En la mayoría de los sistemas web o APIs, esto se considera un rendimiento pobre (el objetivo suele ser < 200ms o incluso < 100ms).
- Upper también alto: 1.28s indica que los peores casos se mantienen cercanos a la mediana. No hay outliers extremos, pero sí una latencia consistentemente elevada.
- Últimos valores similares al promedio: last ≈ mean, por lo que no hay una mejora o empeoramiento significativo reciente. Todo parece estar estable pero todo también es lento.

/exchange (escenario Load con multiples requests)



- *Notese la cantidad de errores que aparecen en las requests cuando la carga es muy alta (50000 req/s).*

Conclusión de los resultados obtenidos

1. Observando los resultados presentados, podemos notar que, en términos de response time, la performance ha empeorado. ¿Por qué ocurre esto?

- Anteriormente, los datos estaban siempre en memoria dentro del servicio; ahora deben ser consultados primero en la base de datos.
- Se introduce la latencia asociada a la conexión con la base de datos. Esto se agrava considerando que utilizamos RedisCloud, lo cual implica más saltos en la red.

2. Por otro lado, en el endpoint más "crítico" POST /exchange vemos una reducción considerable en el uso de CPU y memoria.
3. Finalmente, llama mucho la atención lo performante que es el endpoint PUT /rates. Debería investigarse a fondo la razón de esto.

QAs beneficiados según lo detallado anteriormente:

1. Escalabilidad
2. Disponibilidad
3. Seguridad

QAs perjudicados según lo detallado anteriormente:

1. Performance

2. Escalabilidad Horizontal

Al haber externalizado la base de datos a un servicio independiente del núcleo de arVault, ahora tenemos la posibilidad de escalar horizontalmente el servicio principal. Utilizando NGINX como balanceador de carga, podemos distribuir eficientemente las solicitudes entre múltiples instancias de arVault. Esto no solo mejora la capacidad para manejar un mayor volumen de tráfico, sino que también incrementa la disponibilidad del sistema, eliminando al servicio principal como único punto de falla (SPOF).

Analisis de graficos

Se realizaron pruebas de carga utilizando el endpoint de `POST /exchange` para evaluar el impacto de escalar horizontalmente el servicio `exchange-api` con 3 réplicas, en contraste con su ejecución en una sola instancia. Elegimos este endpoint para realizar estas pruebas ya que fue demostrado anteriormente que es el más exigente a nivel computacional para el servicio.
Aclaración: Como en la lógica de este endpoint se ejecutan sleeps, no necesariamente la performance mejorará con más recursos en términos de tiempo según lo que uno esperaría si no hubieran dichos sleeps.

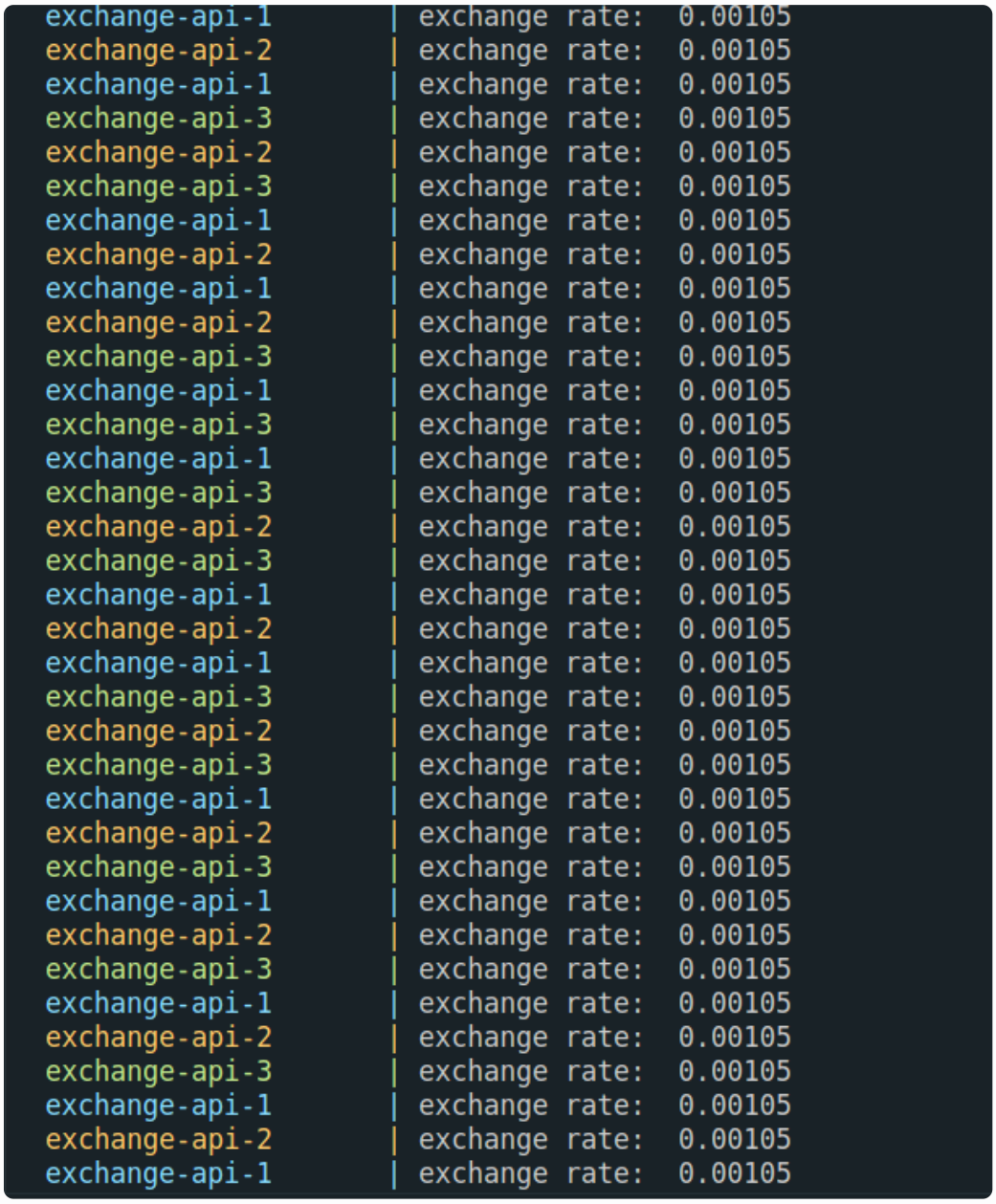
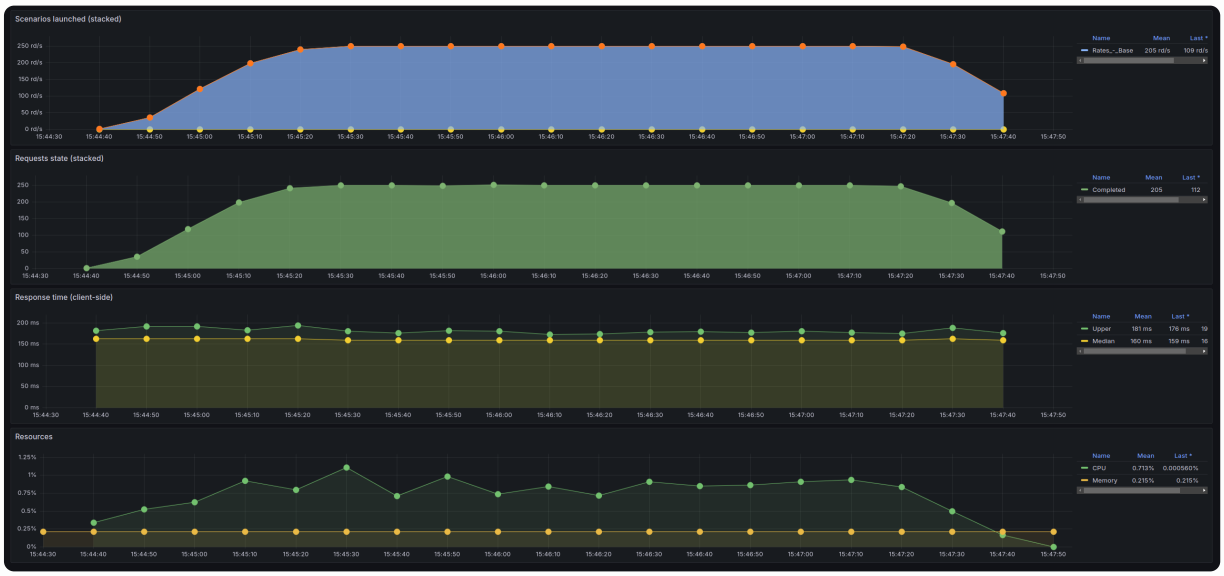


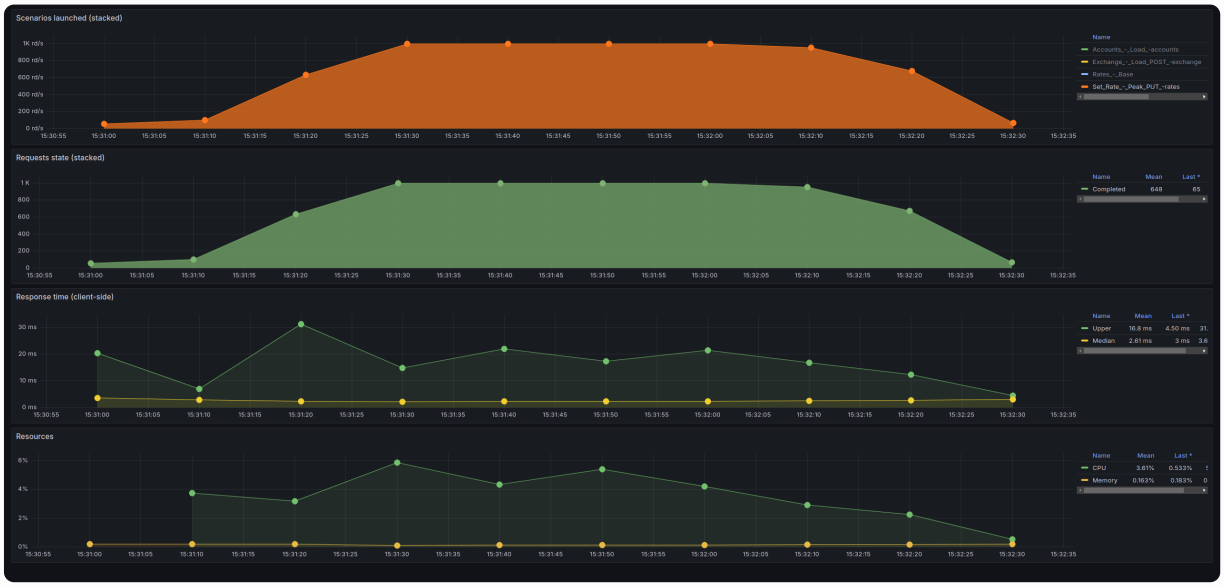
Imagen demostrativa de cómo se balancean las cargas entre las 3 instancias

/accounts (escenario Load)

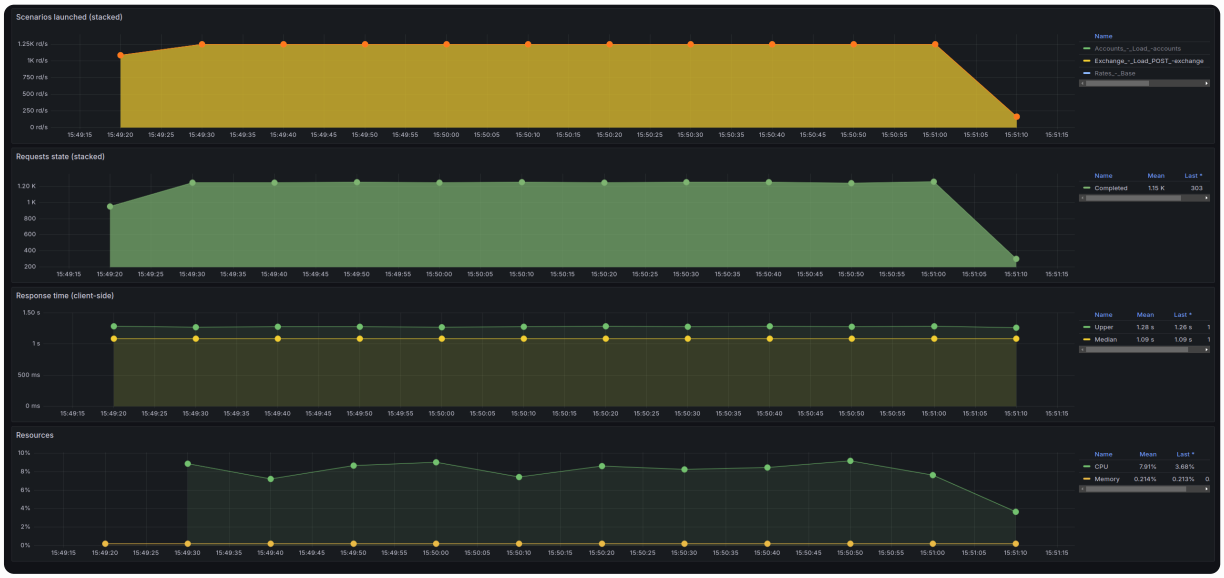
/rates (escenario Base)



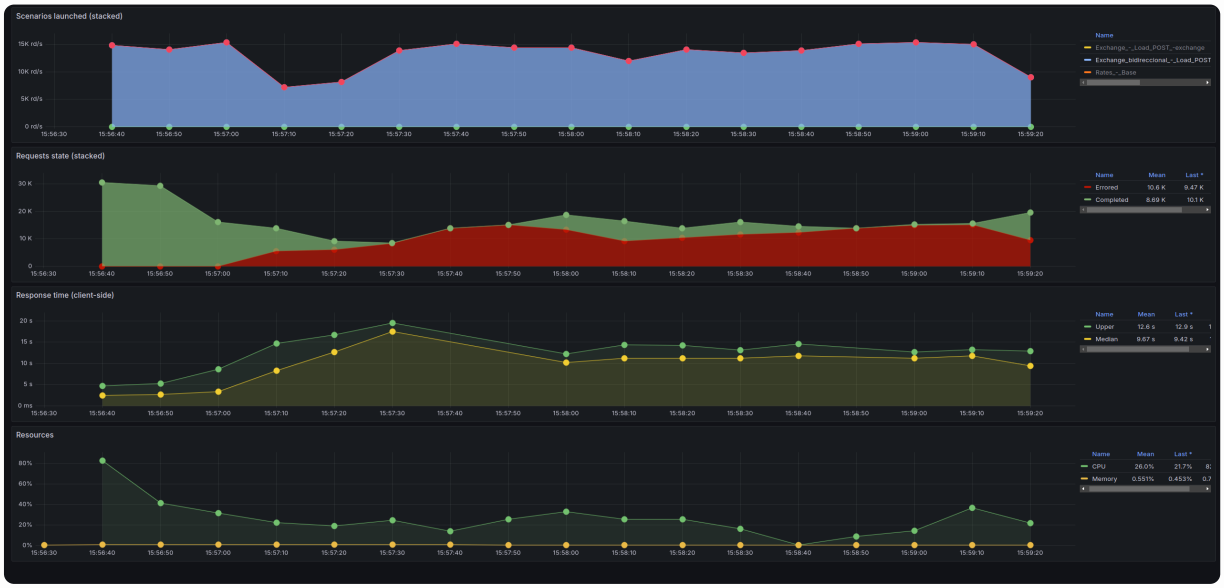
/rates PUT (escenario Peak)



/exchange (escenario Load)



/exchange (escenario Load con multiples requests)



1. Bajo carga normal (arrivalRate: 125)

No se observan diferencias significativas entre tener 1 o 3 réplicas de exchange-api. Tiempos de respuesta, tasa de requests y uso de CPU/memoria son similares en ambos casos.

2. Bajo carga extrema (arrivalRate: 50.000) Ambos escenarios comienzan a degradarse severamente:

- Aparecen muchos errores ECONNRESET, ETIMEDOUT, y http 499 (cliente cierra la conexión antes de que el servidor responda).
- La cantidad de requests exitosas baja abruptamente.

- El uso de CPU puede alcanzar picos cercanos al 90% con 1 réplica.

```
-------------------------------------
Metrics for period to: 15:10:00(-0300) (width: 9.999s)
-------------------------------------

errors.ECONNRESET: ..................................................... 62
errors.ETIMEDOUT: ..................................................... 12574
http.codes.200: ....................................................... 137
http.downloaded_bytes: ................................................ 33102
http.request_rate: .................................................... 1596/sec
http.requests: ........................................................ 15850
http.response_time:
  min: ................................................................ 2405
  max: ................................................................ 9962
  mean: ............................................................... 4856.4
  median: ............................................................. 4316.6
  p95: ................................................................ 9801.2
  p99: ................................................................ 9999.2
http.response_time.2xx:
  min: ................................................................ 2405
  max: ................................................................ 9962
  mean: ............................................................... 4856.4
  median: ............................................................. 4316.6
  p95: ................................................................ 9801.2
  p99: ................................................................ 9999.2
http.responses: ....................................................... 137
vusers.completed: ..................................................... 120
vusers.created: ....................................................... 15838
vusers.created_by_name.Exchange (bidireccional) - Load (POST /exchange): ..... 15838
vusers.failed: ........................................................ 12636
vusers.session_length:
  min: ................................................................ 11787.4
  max: ................................................................ 16178
  mean: ............................................................... 13884.4
  median: ............................................................. 13770.3
  p95: ................................................................ 14917.2
  p99: ................................................................ 15218.6
```

Conclusión de los resultados obtenidos

El escalamiento horizontal a 3 réplicas no muestra beneficios relevantes bajo carga moderada, pero comienza a mostrar ventajas bajo condiciones extremas:

- Mejora la tolerancia a errores (499, ECONNRESET, ETIMEDOUT).
- Distribuye mejor el uso de CPU.
- Aumenta la capacidad del sistema para absorber un volumen más alto de requests sin colapsar inmediatamente.
- Si bien no soluciona por completo el problema de saturación a 50k req/sec, el sistema con múltiples réplicas se degrada más lentamente y con mayor estabilidad que con una sola instancia.

QAs beneficiados según lo detallado anteriormente:

1. Escalabilidad
2. Disponibilidad
3. Performance

QAs perjudicados según lo detallado anteriormente:

1. Simplicidad

3. Rate limiting

Con esta estrategia buscamos demostrar el impacto de aplicar rate limiting en el servicio de backend de la aplicación (sin escalamiento horizontal), particularmente bajo condiciones de carga extrema.

Se diseñó un escenario de prueba con Artillery para estresar el endpoint POST /exchange y observar el comportamiento del sistema bajo diferentes niveles de carga. El objetivo fue generar una sobrecarga suficiente como para provocar fallos y evaluar la estabilidad del sistema sin y con rate limiting.

El servidor está expuesto a través de NGINX, que actúa como proxy inverso hacia el backend (exchange-api). La configuración inicial del servidor no contaba con ningún control de tráfico entrante, lo que permitió la ejecución de pruebas sin restricciones artificiales.

Prueba de artillery ejecutada:

```
config:
  environments:
    api:
      target: "http://localhost:5555"

  phases:
    - name: Warmup
      duration: 10
      arrivalRate: 100
    - name: Spike Up
      duration: 20
      arrivalRate: 300
      rampTo: 1000
    - name: Sustained Fire
      duration: 60
      arrivalRate: 1000
    - name: Aftershock
      duration: 30
      arrivalRate: 1000
      rampTo: 200
    - name: Cooldown
      duration: 20
      arrivalRate: 200
      rampTo: 0

  http:
    pool: 2000
    timeout: 10
    maxSockets: 5000

  defaults:
    headers:
```

```
      content-type: application/json

scenarios:
  - name: Overkill (POST /exchange)
    flow:
      - loop:
          - post:
              url: "/exchange"
              json:
                baseCurrency: "ARS"
                counterCurrency: "EUR"
                baseAccountId: 1
                counterAccountId: 3
                baseAmount: 10
        count: 100
```

A continuación se muestran los resultados que arroja la prueba de artillery sin rate limiting:

```
--------------------------------
Summary report @ 17:24:08(-0300)
--------------------------------

errors.ECONNRESET: ..................................................... 11183
errors.ETIMEDOUT: ...................................................... 81200
http.codes.200: ........................................................ 197059
http.codes.502: ........................................................ 659
http.downloaded_bytes: ................................................. 47594682
http.request_rate: ..................................................... 1008/sec
http.requests: ......................................................... 290101
http.response_time:
  min: ................................................................. 7
  max: ................................................................. 10653
  mean: ................................................................ 1982.7
  median: .............................................................. 1300.1
  p95: ................................................................. 5378.9
  p99: ................................................................. 7709.8
http.response_time.2xx:
  min: ................................................................. 887
  max: ................................................................. 10653
  mean: ................................................................ 1989.2
  median: .............................................................. 1300.1
  p95: ................................................................. 5378.9
  p99: ................................................................. 7709.8
http.response_time.5xx:
  min: ................................................................. 7
  max: ................................................................. 370
  mean: ................................................................ 57.8
  median: .............................................................. 46.1
  p95: ................................................................. 144
  p99: ................................................................. 252.2
http.responses: ........................................................ 197718
vusers.completed: ...................................................... 1601
vusers.created: ........................................................ 93984
vusers.created_by_name.Overkill (POST /exchange): ...................... 93984
vusers.failed: ......................................................... 92383
vusers.session_length:
  min: ................................................................. 160276.5
  max: ................................................................. 206573.5
  mean: ................................................................ 181612.3
  median: .............................................................. 181743.9
  p95: ................................................................. 196881.3
  p99: ................................................................. 200858.7
```

Notese la cantidad de solicitudes fallidas debido a ECONNRESET, ETIMEDOUT, o 502.

Resultados con rate limiting en NGINX

Aplicando la siguiente configuración de NGINX:

```
limit_req_zone $binary_remote_addr zone=apiLimit:10m rate=100r/s;

location / {
    limit_req zone=apiLimit burst=50 nodelay;
    proxy_pass http://api/;
}
```

Y ejecutando la misma prueba de estrés, recibimos las siguientes estadísticas:

```
-------------------------------
Summary report @ 17:36:25(-0300)
-------------------------------

errors.ECONNRESET: ..... 5056
errors.ETIMEDOUT: ..... 88376
http.codes.200: ..... 3076
http.codes.503: ..... 97812
http.downloaded_bytes: ..... 20010280
http.request_rate: ..... 240/sec
http.requests: ..... 194320
http.response_time:
  min: ..... 0
  max: ..... 10492
  mean: ..... 1121.8
  median: ..... 83.9
  p95: ..... 6439.7
  p99: ..... 7865.6
http.response_time.2xx:
  min: ..... 893
  max: ..... 9574
  mean: ..... 3368.6
  median: ..... 2276.1
  p95: ..... 8520.7
  p99: ..... 9416.8
http.response_time.5xx:
  min: ..... 0
  max: ..... 10492
  mean: ..... 1051.2
  median: ..... 80.6
  p95: ..... 6312.2
  p99: ..... 7865.6
http.responses: ..... 100888
vusers.completed: ..... 552
vusers.created: ..... 93984
vusers.created_by_name.Overkill (POST /exchange): ..... 93984
vusers.failed: ..... 93432
vusers.session_length:
  min: ..... 1357.7
  max: ..... 100984
  mean: ..... 63902
  median: ..... 64236
  p95: ..... 80043.6
  p99: ..... 92072.4
```

Conclusión de los resultados obtenidos

Sin Rate Limiting:

- **Número de respuestas 200**: 197,059 respuestas exitosas (código 200).
- **Errores 503**: No se registraron.
- **Errores de conexión y tiempo de espera**: Se registraron 11,183 errores de conexión (ECONNRESET) y 81,200 errores de tiempo de espera (ETIMEDOUT).

- **Tasa de solicitudes**: 1008 solicitudes por segundo, lo que sugiere que el servidor pudo manejar el tráfico con un buen rendimiento.

Con Rate Limiting:

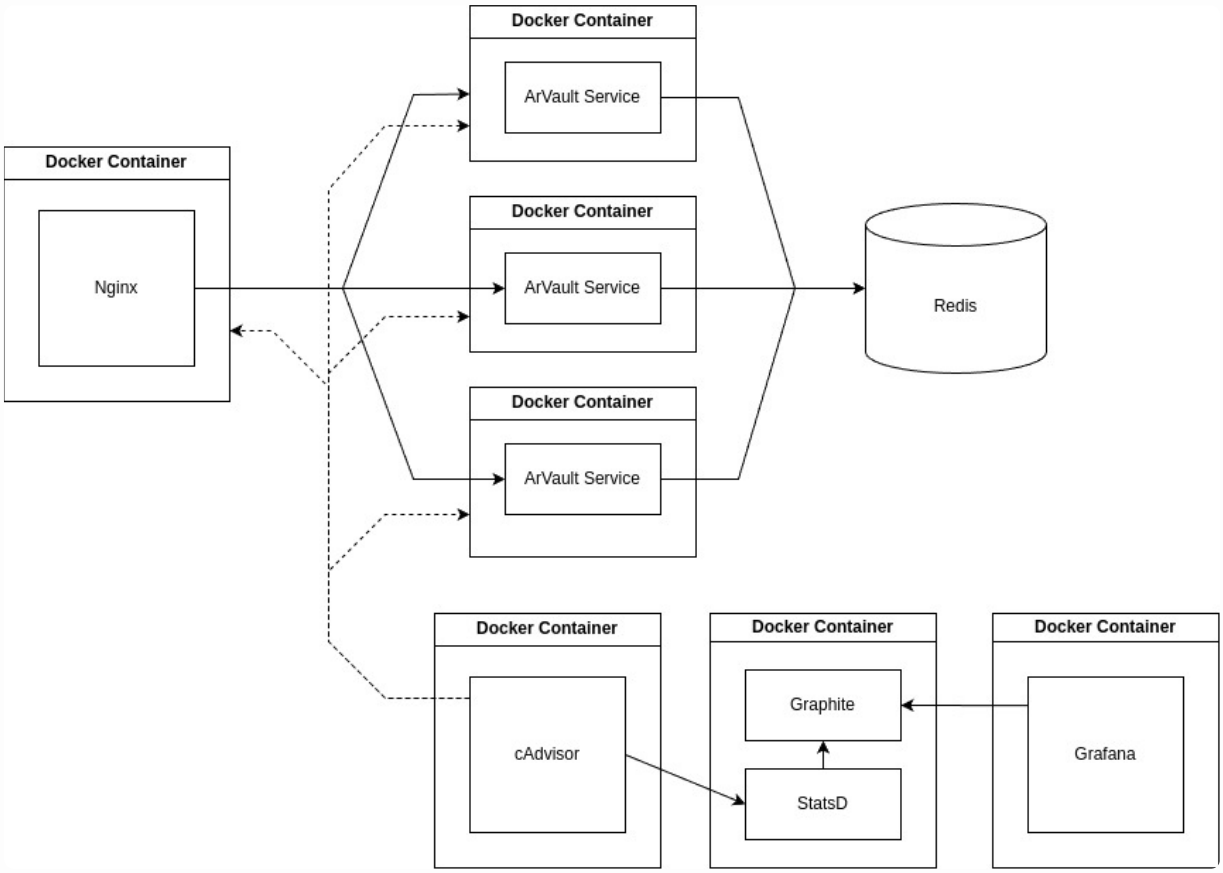
- **Número de respuestas 200**: 3,076 respuestas exitosas (código 200), lo que representa solo el 1.56% de las solicitudes procesadas comparado con la prueba sin rate limiting.
- **Errores 503**: 97,812 errores 503, lo que representa aproximadamente el 50% de las solicitudes rechazadas debido al rate limiting.
- **Errores de conexión y tiempo de espera**: Los errores ETIMEDOUT permanecieron altos (88,376), pero los errores ECONNRESET se redujeron en casi el 50%.
- **Tasa de solicitudes**: La tasa de solicitudes disminuyó significativamente a 240 solicitudes por segundo debido a que muchas solicitudes fueron bloqueadas por el rate limiting.

1. **Rate Limiting**: El rate limiting está funcionando correctamente, pero está limitando severamente la capacidad de procesar solicitudes. Muchas solicitudes están siendo rechazadas con el código 503 porque exceden el límite de solicitudes por segundo. En caso de esperar un caudal tan grande de solicitudes, deberíamos permitir límites más flexibles para evitar rechazar tantas requests. Caso contrario de que no sea un flujo normal, quizás es correcto dejar límites tan justos para evitar ataques.

QAs beneficiados según lo detallado anteriormente:

1. Disponibilidad
2. Seguridad
3. Performance (no comprobado, pero debería ya que ajusta la cantidad de requests recibidos a la cantidad de recursos disponibles)

Diagrama de componentes de arquitectura evolucionada



Estrategias no utilizadas

1. Cachear Recursos Comunmente Solicitados

Dado a que la base de datos es ya una caché, y que la data solicitada es tal cual la almacenada en la base de datos (es decir, no se le aplica un procesamiento extra en el servicio que la modifique o convierta), creemos que no tiene sentido agregar una caché porque los beneficios de esto no compensan la inversión.

Métricas Propias - Volumen Operado por Moneda (Total y Neto)

Se agregaron métricas personalizadas para medir el volumen operado por moneda, tanto en forma total como neta, en el endpoint `/exchange`.

Estas métricas se envían a StatsD al completar exitosamente una operación de cambio y se visualizan en el dashboard de Grafana bajo el panel **“Volumen Operado por Moneda (Total y Neto)”**.

Implementación técnica

Las métricas registradas son las siguientes:

```
statsd.increment(`volume.total.${baseCurrency}`, Math.round(baseAmount * 100));
statsd.increment(`volume.net.${baseCurrency}`, Math.round(-baseAmount * 100));
statsd.increment(`volume.net.${counterCurrency}`, Math.round(counterAmount * 100)
```

Cada valor se multiplica por 100 para registrar los montos en centavos y evitar pérdida de precisión en los acumuladores de StatsD.

Métricas registradas

Métrica	Descripción
volume.total.ARS	Total acumulado de ARS entregados (base currency en las operaciones de cambio).
volume.net.ARS	Diferencia neta de ARS: positivo si se adquirió ARS, negativo si se entregó.
volume.total.USD	Total acumulado de USD adquiridos (counter currency en las operaciones de cambio).
volume.net.USD	Diferencia neta de USD: positivo si se recibió USD, negativo si se vendió.

Escenario de carga bidireccional

Para demostrar de forma más clara cómo las métricas `volume.total` y `volume.net` reflejan el comportamiento real del sistema, se utilizó un escenario de carga con operaciones bidireccionales.

Objetivo

Este enfoque permite simular tanto **ventas** como **compras** de cada moneda, generando así:

- Cambios **acumulativos** en el volumen total operado,
- Variaciones **positivas y negativas** en el volumen neto, reflejando los flujos reales.

Configuración utilizada

```
scenarios:
  - name: Exchange (bidireccional) - Load (POST /exchange)
    flow:
```

```
    - post:
        url: "/exchange"
        json:
          baseCurrency: "ARS"
          counterCurrency: "USD"
          baseAccountId: "1"
          counterAccountId: "2"
          baseAmount: 10000
    - post:
        url: "/exchange"
        json:
          baseCurrency: "USD"
          counterCurrency: "ARS"
          baseAccountId: "2"
          counterAccountId: "1"
          baseAmount: 10
```

Primera operación:

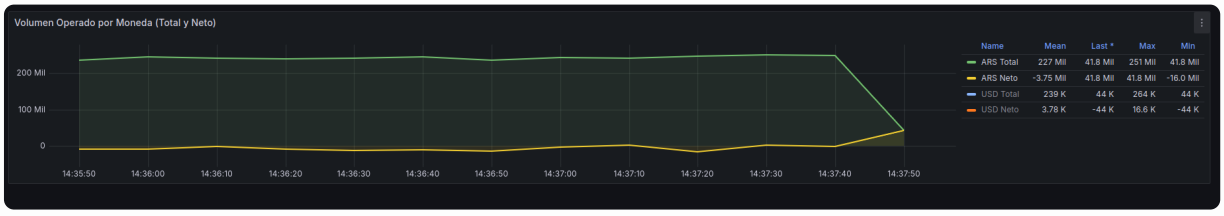
- Vende 10.000 ARS (cuenta 1) para adquirir USD (cuenta 2),
- Impacta negativamente en el neto de ARS y positivamente en el neto de USD.

Segunda operación:

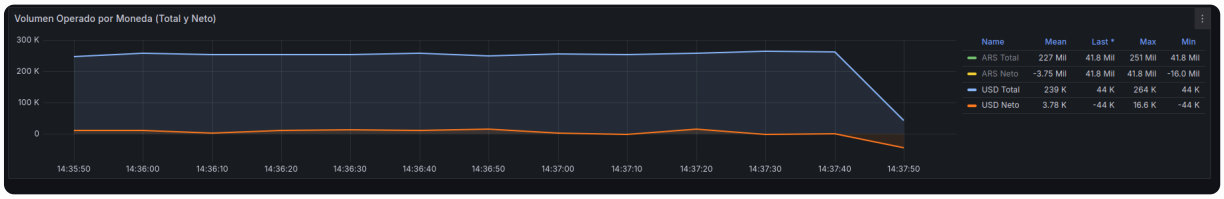
- Vende 10 USD (cuenta 2) para adquirir ARS (cuenta 1),
- Impacta negativamente en el neto de USD y positivamente en el neto de ARS.

Visualización en Grafana

ARS - Total y Neto



USD - Total y Neto



Análisis de resultados

Balances antes del escenario de carga:


```
[
  { "id": 1, "currency": "ARS", "balance": 50000 },
  { "id": 2, "currency": "USD", "balance": 3000 }
]
```


Balances después del escenario de carga:


```
[
  { "id": 1, "currency": "ARS", "balance": 21580 },
  { "id": 2, "currency": "USD", "balance": 3031.5 }
]
```


Esto indica que:

- Se utilizaron **~28.420 ARS** (`50k - 21.58k`),
- Se ganaron **~31.5 USD** (`3031.5 - 3000`),
- Las métricas `volume.total` y `volume.net` reflejan correctamente estas operaciones:
  - ARS Total ≈ 227.000 centavos (sumando ida y vuelta),
  - USD Total ≈ 239.000 centavos,
  - ARS Neto ≈ -3.750 centavos,
  - USD Neto ≈ +3.780 centavos.

Conclusión

Los gráficos permiten visualizar en tiempo real el volumen de operaciones ejecutadas por moneda, tanto acumulado como neto.
Esto cumple con el requisito funcional obligatorio del cliente:

> “[OBLIGATORIO] Agregar métricas que muestren el volumen operado en cada moneda (compras y ventas sumadas por moneda), como así también el neto (compras suman y ventas restan), ambos a medida que transcurre el tiempo.”

Estas métricas fueron validadas cruzando los resultados visuales en Grafana con los saldos previos y posteriores de las cuentas involucradas, comprobando su consistencia.