Arquitectura de Software - TB034

arVault

Informe Trabajo Práctico 1

Grupo 2 - Los Simuladores

Alumnos:

Marco Natalini - 108056

Ignacio Rodriguez Justo - 108267

Joaquin Dopazo - 107436

Francisco Rovira Rossel - 107536

2C 2025 - Facultad de Ingeniería - Universidad de Buenos Aires

Análisis inicial

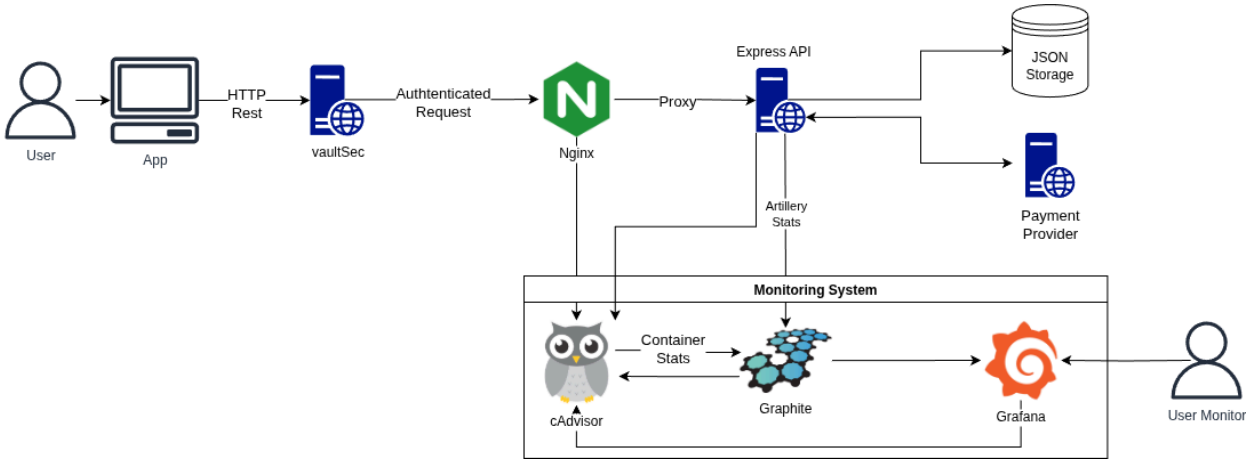
Análisis de Arquitectura

La arquitectura inicial del proyecto arVault viene dada de la siguiente forma:

- El **servicio principal**, construido en NodeJS + Express, escucha las peticiones que le llegan del **Reverse Proxy** (o Load Balancer), exponiendo distintos endpoints para su interacción, que hacen las siguientes operaciones:
    - **/rates**: Devuelve la información de los tipos de cambio que hay entre las monedas y el Peso Argentino, también se pueden actualizar los valores para los distintos tipos de cambio
    - **/accounts**: Devuelve la información de las cuentas internas de la empresa (su id, balance y tipo de moneda), así como la posibilidad de consultar el balance para una cuenta en particular
    - **/log**: Devuelve las operaciones que fueron registradas a lo largo del funcionamiento de la aplicación
    - **/exchange**: Se utiliza para realizar transferencias de dinero entre las cuentas de los clientes
- El **almacenamiento** de la información (tanto de los logs como el estado de las cuentas y tipos de cambio) se hace a través de archivos .json que se guardan localmente y se reinician después de cada corrida.
- Tanto la **autenticación** como el **manejo de las transferencias** es manejado con un servicio externo, entonces no los incluimos en el análisis como elementos clave para tomar decisiones que impacten en los QA.

Diagrama Componentes y Conectores (caso base)

Para el caso base se armó el siguiente diagrama de Componentes y Conectores:



En este diagrama se puede mostrar el flujo para una request de ejemplo:

- Un usuario a través de la aplicación hace una solicitud a nuestro servidor
- Ésta primero pasa por un servicio de autenticación tercerizado (vaultSec) y luego llega al nodo Nginx (API Gateway, load balancer, reverse proxy o como queramos referirnos a este)
- Luego la solicitud llega realmente a nuestro servidor, haciendo el procesamiento correspondiente.
- Este procesamiento queda registrado por las herramientas utilizadas (Graphite, CAdvisor, Grafana) para que después puedan ser visualizadas, por ejemplo, por un Analista/Administrador (en este caso, nosotros haciendo el monitoreo de las métricas)
- En caso de ser necesario, se modifican los archivos .json correspondientes, y en caso de que sea una transferencia se simula un pedido a un agente de pagos para procesar la transferencia.

Análisis de Infraestructura

La infraestructura base que se adjunto para este Trabajo Práctico consiste en:

- Un nodo destinado a la aplicación en sí, que no se puede acceder directamente sino a través del servicio explicado a continuación.
- Un nodo **nginx** destinado a ser el intermediario (o reverse proxy) entre el cliente haciendo las peticiones y el servidor *app*, escuchando pedidos a través del puerto 5555 localmente y mapeando las conexiones al puerto 80.

- Un nodo **graphite** destinado a recolectar y almacenar métricas del servicio
- Un nodo **grafana** utilizado para visualizar las métricas brindadas por graphite, mostrando distintos dashboards según los datos que nos interesen, se puede acceder al dashboard principal gracias a que está expuesto el puerto 80.
- Finalmente, un nodo **cadvisor** que es el que provee las métricas a graphite

Todo esto corre de forma virtualizada a través de **contenedores de Docker**, facilitando el despliegue en cualquier máquina

Análisis de código

El código brindado en esta etapa inicial consta de:

- Los paquetes necesarios para que la aplicación corra, listados en *package.json.*
- Un Dockerfile que contiene las instrucciones necesarias para que el servicio **app** quede correctamente dockerizado.
- Documentación de la aplicación, escrita por el dueño del producto, que contiene explicación muy básica del funcionamiento así como de los componentes principales y faltantes, y desgloses sobre configuración, almacenamiento, endpoints, cuentas, exchanges y logs.
- Snippets de código:
    - **app.js**: Punto de entrada de la aplicación, expone los principales endpoints (**account, rates, log, exchange**). Cada uno extrae los campos principales del body, hace una validación muy básica de campos y llama a las principales funciones que se encuentran en **exchanges.js**. Se busca cambiar este comportamiento para que las validaciones sean más robustas. Por ejemplo, accounts
    - **exchange.js**: Se encarga de la lógica de negocio en la aplicación. Maneja el cálculo de los tipos de cambio y los movimientos financieros entre cuentas. Se menciona en el README de la app que la lógica de las cuentas y los movimientos está muy acoplada y que se debería hacer un servicio aparte para el manejo de las cuentas. Además hay una clara falta de validación de parámetros, por ejemplo al tratar de operar con monedas que no se encuentran en la aplicación, esta responde con un código de error inapropiado (500) y colapsa

completamente. Otro ejemplo es que solo el peso tiene valor recíproco con otras monedas, intentar cambiar dólares a euros también provoca errores, pero estos no causan la caída del servicio.

- **state.js**: Es la capa de persistencia/repositorio de la aplicación. Guarda la información de las cuentas en archivos *JSON* ubicados en *app/state/* y también se encarga de loguear las operaciones en cada intervalo de tiempo de 5 segundos. El README también menciona que lo mejor es deshacernos de estos archivos .json para las cuentas, y que estén almacenados de otra forma, por ejemplo, usando una base de datos.

QA claves

Los Atributos de Calidad comprometidos de esta aplicación son:

- **Availability:** El problema principal que tuvo la gente de arVault fue: *“problemas en el uso de la función de cambio de monedas”*, indicando una escasez de este atributo de calidad. Esto puede indicar una falta de respuesta del servidor ante pedidos de los usuarios, degradando notablemente el rendimiento ante grandes volúmenes de requests. Esto sumado a que actualmente está corriendo una sola instancia del servidor, lo hace más propenso a no tener un nivel aceptable de disponibilidad.
  Tampoco se ve en el código inicial que haya medidas robustas para la detección de fallas y para la recuperación del servicio ante las mismas, quedando vulnerable a que un error provoque una ola de usuarios insatisfechos. Medidas como introducir rate limiting en el nodo de nginx pueden llegar a mejorar este atributo.
- **Performance:** Los antecedentes mencionan que en la ronda inicial de inversión, los capitales fueron destinados al agregado de features en vez de al fortalecimiento de la arquitectura e infraestructura, esto tiene un impacto directo en la performance ya que ahora el sistema tiene que hacer más cosas con las mismas herramientas. Si bien es bastante simple la implementación inicial, nos llamó la atención un par de cosas del código en sí:

- Primero, una línea en particular del código, encontrada en la función *transfer* en el archivo *exchange.js*:

  ```
  return new Promise((resolve) => setTimeout(() => resolve(true), Math.random() * (max - min + 1) + min));
  ```

  Esto fue aclarado que fue hecho con la intención de simplificar el código y juega el papel de una comunicación con un servicio externo para las transferencias.
- Sin embargo, esta función se utiliza en la función *exchange* de la siguiente forma:
  - Primero para transferir fondos para un tipo de moneda entre una cuenta origen de un cliente y una cuenta destino interna de la empresa
  - Segundo, para transferir fondos en otro tipo de moneda entre una cuenta origen interna de la empresa hacia una cuenta destino de un cliente
  - Si falla el segundo paso, se hace la transferencia del paso 1 pero a la inversa

  Se puede ver que esto tiene pasos innecesarios (sobre todo el tercero) que se pueden solucionar haciendo previamente un chequeo de saldos (asumimos que en el paso 1 el cliente ya tiene los fondos suficientes aunque es dudoso para la integridad de las transacciones en un escenario real)

- Por último, el grabado de los logs cada 5 segundos puede llegar a ser una decisión cuestionable, esto será algo que se decida cambiar en base al análisis de las métricas iniciales del sistema. Hay arreglos rápidos y fáciles como por ejemplo no sobreescribir el archivo siempre, sino agregar lo que falta.

Observando la arquitectura y decisiones sobre los componentes utilizados, el nginx por ahora no tiene funciones que aporten valor como reverse proxy (no está balanceando carga, los aspectos de seguridad ya están siendo tercerizados, no hay rate limiting para que no desborde de consultas al servicio principal, etc.). Esto puede tener un impacto en la performance al

añadir un paso extra en cada solicitud en este estado inicial donde hay 1 solo servidor

- **Scalability:** En relación con Availability, todo apunta a que el sistema no escala bien cuando la base de usuarios va en crecimiento. Un problema claro es que hay un solo nodo en el sistema, impidiendo escalabilidad horizontal, y también haciendo que el nginx no tenga un rol de distribución de carga y este siendo un elemento intermedio por el que se pasa agregando latencia. Una solución es replicar horizontalmente estos servidores y que el nodo nginx tenga un rol de distribuidor de consultas (teniendo en cuenta aspectos como saldos de cuentas por ejemplo, aunque eso involucra complejidad si se quiere hacer que todos los servidores compartan la misma información, sino hay más riesgo de que consultas sean redirigidas a servidores que no están aptos para realizar operaciones de exchange).
- **Security:** Este atributo de calidad se encuentra tercerizado y se asume que se cumple. Sin embargo pueden implementarse medidas más robustas en cuanto a la validación de datos de entrada por ejemplo, y de también de chequeo de balances relacionado con lo hablado en la sección de *Performance*.
- **Maintainability:** Por el estado inicial del proyecto, la mantenibilidad es simple teniendo en cuenta las recomendaciones del dueño del proyecto, como abstraer el manejo de cuentas (ahora esta lógica está junto con la de los tipos de cambio) y el uso de una base de datos. Por lo demás, no es un atributo que se haya descuidado tanto como anteriores. Lo que hay que procurar es no degradar mucho este atributo de calidad a la hora de implementar mejoras en los otros (por ejemplo: replicación de servidores con el guardado inicial puede hacer de la redistribución de carga algo que induzca fallos, que puede ser mitigado por ejemplo con una base de datos que centralice la información de los montos de las cuentas internas de la empresa).
- **Testability:** En el estado inicial el proyecto posee la increíble cantidad de 0 (cero) tests que verifiquen su funcionamiento correcto para determinados escenarios. Esto no indica que el sistema no sea testeable, sino que es un atributo que al descuidarlo puede provocar mal funcionamiento de los componentes a futuro. La medición de este atributo puede inducir a

conclusiones erróneas (tener tests no es una condición suficiente para asegurar Testabilidad), entonces nos limitamos a analizar este atributo de una forma más teórica, ya que formas de implementar pruebas hay muchas y creemos mejor dedicar esfuerzos a mejorar atributos que impactan en las métricas visibles para un usuario

- **Usabilidad:** Al ser código de backend lo que estamos desmenuzando, hablar de este atributo en cara a la experiencia de usuario es un poco difícil ya que este no interactúa directamente con nosotros sino con una UI dedicada. Dicho esto, el sistema es lo suficientemente simple como para decir que este atributo no está siendo un protagonista esencial en este análisis, pero la arquitectura inicial y su despliegue en Docker hacen muy maleable todos sus aspectos, así que tampoco está muy dejado de lado. Lo que sí nos llama la atención es todo lo hablado en la sección de *Performance* que pueda tener un impacto en la usabilidad, como lo es los tiempos muertos al querer hacer una operación de cambio de monedas.
- **Interoperability:** En cuanto al armado del servicio principal de la aplicación, las capas existentes (app, exchange, state) tienen interfaces definidas que se respetan a lo largo de la ejecución del programa, puede ser mejorado si el estado se maneja a través de una base de datos en vez de un par de archivos json, y si se logra separar la lógica de las cuentas con los fondos para hacer los intercambios de monedas.

Análisis y gráficos de métricas

Para la siguiente sección se van a proveer gráficos de rendimiento para el principal caso de uso problemático de este proyecto. Los escenarios a probar son los siguientes:

- **Ramp up:** El proyecto vino con un escenario simple, mostrado en este informe.
- **Pico:** Un pico de pedidos, de ahora en más las operaciones hechas van a estar relacionadas con el intercambio de monedas.
- **Exponencial:** Prueba de crecida exponencial de pedidos.
- **Intermitente:** Periodos manejables de intensidad y normalidad.
- **Stress:** Escenario de creciente intensidad para observar uso de recursos y en particular el uso de disco por parte del servicio principal.

A partir de estos escenarios se van a sacar conclusiones sobre el caso base teniendo en cuenta el previo análisis de los atributos de calidad, y se van a proponer mejoras que creemos que pueden potenciar los atributos más comprometidos.

Caso 1: Ramp-up

Estas métricas introductorias ilustran un caso muy simple usando el endpoint **/rates**, el cual devuelve una información constante (siempre y cuando no se cambie, que no es el caso), con un volumen de solicitudes también bajo. Con esto damos pie a la interpretación de los gráficos, como se puede ver el uso de recursos aumenta de acuerdo a la llegada de requests, los tiempos medios de respuestas no se ven

afectados gravemente, y el sistema maneja tranquilamente este volumen. En cuanto a los valores netos y movimientos, no hay datos para mostrar ya que no se modificaron estos valores.

Los escenarios posteriores van a evaluar la parte problemática de los usuarios (intercambio de monedas) junto con un aumento considerable del volumen de solicitudes.

Caso 2: Pico de solicitudes

Estas métricas tienen como objetivo evaluar la resiliencia del sistema ante picos abruptos de tráfico. Como podemos observar se ve una curva triangular con un pico alto de requests y una caída igualmente rápida.

Durante el ascenso y en el pico aparecen zonas rojas (errores) ,esto indica que al superar un umbral de carga, algunas requests no pueden completarse correctamente, demostrando problemas de disponibilidad cuando hay un aumento repentino de requests en el servidor. La latencia sube abruptamente durante el pico (coincide con los errores) y luego se normaliza, esto evidencia que el sistema degrada su performance bajo estrés. El uso de CPU también muestra un incremento pronunciado, alcanzando su punto máximo durante el pico y cayendo después. El volumen (compras + ventas) sube de forma brusca en el momento del pico y luego cae a cero, esto significa que el sistema procesa correctamente gran parte de las operaciones antes del límite de saturación. Los gráficos referentes a los volúmenes operados y valores netos de cada moneda representan la unidireccionalidad de los

movimientos (repitiendose para el resto de casos). Puede parecer que solo la moneda ARS se ve afectada, pero no se llega a apreciar en el gráfico el movimiento de la otra moneda (USD), ya que la cantidad crece muy poco con respecto a la otra (como referencia, el tipo de cambio que se estableció para estos escenarios es 1 USD = 1500 ARS).

Caso 3: Exponencial

En este caso de prueba, a medida que avanza el tiempo las solicitudes aumentan de manera exponencial.

Notamos que en un principio las request no tienen problema en ejecutarse, pero a medida que aumentan la cantidad, el servidor se satura, comenzando a fallar cuando llega a recibir cerca de 1000 requests por segundo, esto es un claro problema con la escalabilidad de usuarios. Además podemos observar que la latencia de respuestas también pasa de ser baja a muy alta a la vez que el servidor se encuentra saturado, afectando la performance del sistema. La CPU muestra un patrón de crecimiento acompañado de picos: refleja el esfuerzo del servidor en procesar operaciones más densas, teniendo un bajón en el mismo periodo donde aparecen los errores, siendo síntoma de la imposibilidad de satisfacer los pedidos. El volumen operado (compras + ventas) crece con la carga, alcanzando un máximo y luego cayendo cuando aparecen errores en el procesamiento de las transacciones.

Caso 4: Intermitente

El objetivo de esta prueba no era llevar el servicio al límite, sino de analizar si hay varianza en el tiempo de respuesta ante picos consistentes de carga y descarga

El gráfico muestra un patrón en forma de picos pequeños, característico de los bursts periódicos. Todas las solicitudes aparecen en verde (completadas con éxito). No se observan errores ni tiempos de espera prolongados, lo que indica que el sistema maneja bien cargas intermitentes y recupera recursos entre las ráfagas. Las latencias se mantienen constantes y bajas durante todo el ciclo, sin picos ni degradación acumulada, esto evidencia una buena estabilidad del servidor bajo cargas intermitentes. La CPU refleja un patrón oscilante en sincronía con los bursts. El volumen de monedas fluctúa de manera proporcional a cada ráfaga. El neto (compras - ventas) mantiene un comportamiento estable, sin desviaciones fuertes, lo que indica consistencia en las operaciones aún bajo carga cíclica.

Caso 5: Stress

El objetivo de estas métricas es determinar el punto de saturación del sistema y analizar su comportamiento cuando la carga supera su capacidad de procesamiento. En este escenario las solicitudes aumentan progresivamente hasta alcanzar valores muy altos, con un crecimiento escalonado que busca forzar los límites del servicio.

Podemos observar que, al comienzo, las requests se ejecutan correctamente y las respuestas mantienen una latencia baja. Sin embargo, a medida que la carga aumenta, las solicitudes completadas comienzan a disminuir y aparecen zonas rojas en el gráfico de estado, indicando una cantidad creciente de errores. Esto marca el punto en el que el servidor deja de poder manejar todas las operaciones simultáneamente. En comparación con el test exponencial podemos notar durante un plazo mayor de tiempo que luego de que el servidor se satura, no vuelve a recuperarse.

La latencia de respuesta también se incrementa de forma notoria cuando el sistema se encuentra saturado, pasando de tiempos estables a valores que superan los varios segundos por request. En cuanto a los recursos, vemos que el procesamiento supera el uso de un solo núcleo teniendo una media bastante elevada con respecto a escenarios anteriores, mientras que la memoria crece lentamente hasta estabilizarse. Además, la nueva métrica de uso de disco muestra un crecimiento del

espacio ocupado principalmente por los archivos de log y estado, confirmando que la persistencia en archivos JSON es uno de los cuellos de botella del sistema bajo carga extrema.

El volumen operado por moneda (compras + ventas) crece con la carga hasta que los errores comienzan a aparecer, momento en el que desciende bruscamente. El neto por moneda (compras − ventas) se mantiene positivo pero deja de variar, lo que sugiere que muchas transacciones no alcanzaron a completarse correctamente.

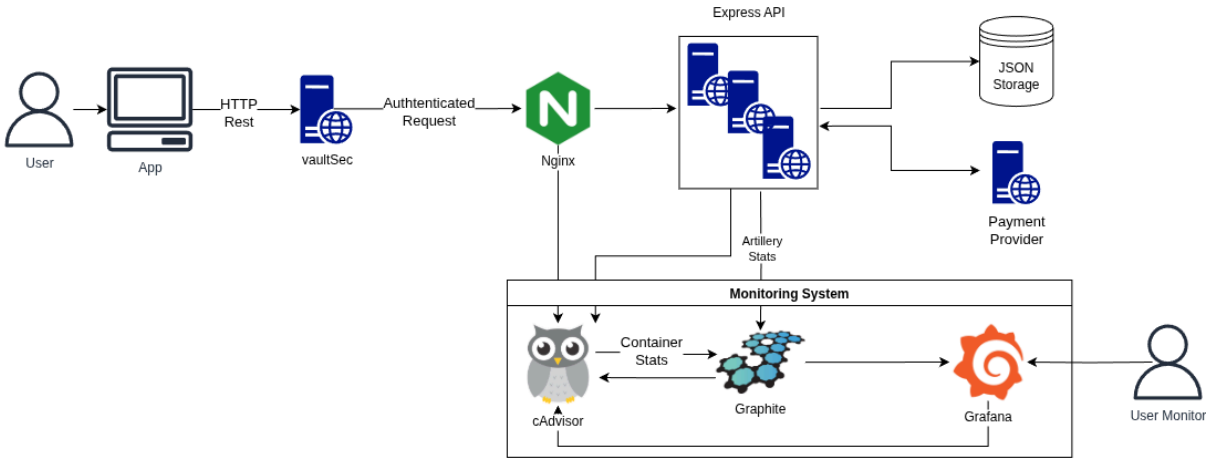
Ya con estos escenarios estudiados tenemos la suficiente información como para empezar a aplicar tácticas para poder manejar los casos problemáticos.

Tácticas a aplicar

Diferenciamos las tácticas efectivamente aplicadas de aquellas que aportarían una mejoría pero no implementamos, aunque discutimos su utilidad e hipotético impacto en distintas situaciones:

Replicación

La primera técnica que aplicamos fue la **Replicación**, esto es, añadir instancias separadas del servicio principal para que el tráfico se redistribuya y que esto pueda ayudar a mitigar los problemas. Adjuntamos el diagrama de Componentes y Conectores para este caso



En este caso se replica la Express API, se adapta Nginx para funcionar como “Load Balancer”, distribuyendo los pedidos a las distintas instancias.

Caso 1: Ramp-up (Replicación)

La única diferencia destacable para este caso es que el tiempo de respuesta parece haber aumentado considerablemente. Posiblemente la causa se debe al tiempo que tarda el Load Balancer en asignar la tarea a una de las réplicas y luego en escuchar y devolver el resultado.

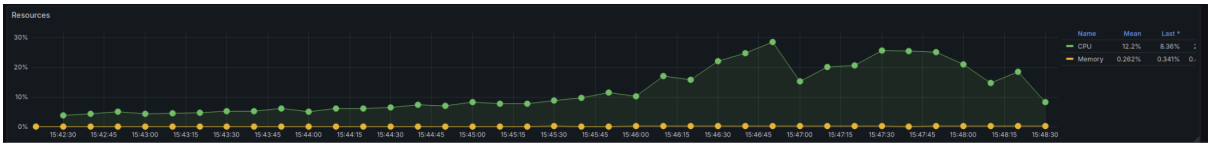
Caso 2: Spike (Replicación)

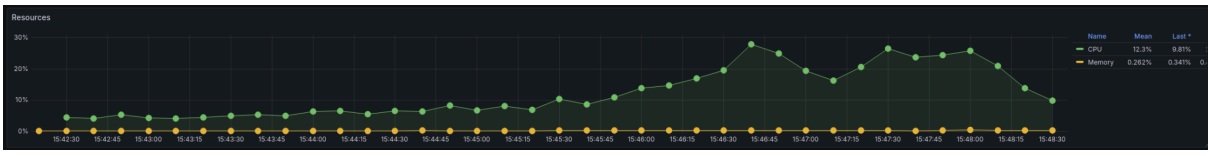
En este caso sí se nota un cambio significativo pero es uno negativo, el sistema no logró contestar la misma cantidad de solicitudes que la vez pasada, fallando la inmensa mayoría.

Caso 3: Exponencial (Replicación)

No se puede notar un impacto positivo notable. La cantidad de solicitudes que fallaron es muy ligeramente menor, pero el tiempo de respuesta es idéntico, y no se consiguió que el sistema tolerase una mayor cantidad de solicitudes simultáneas.

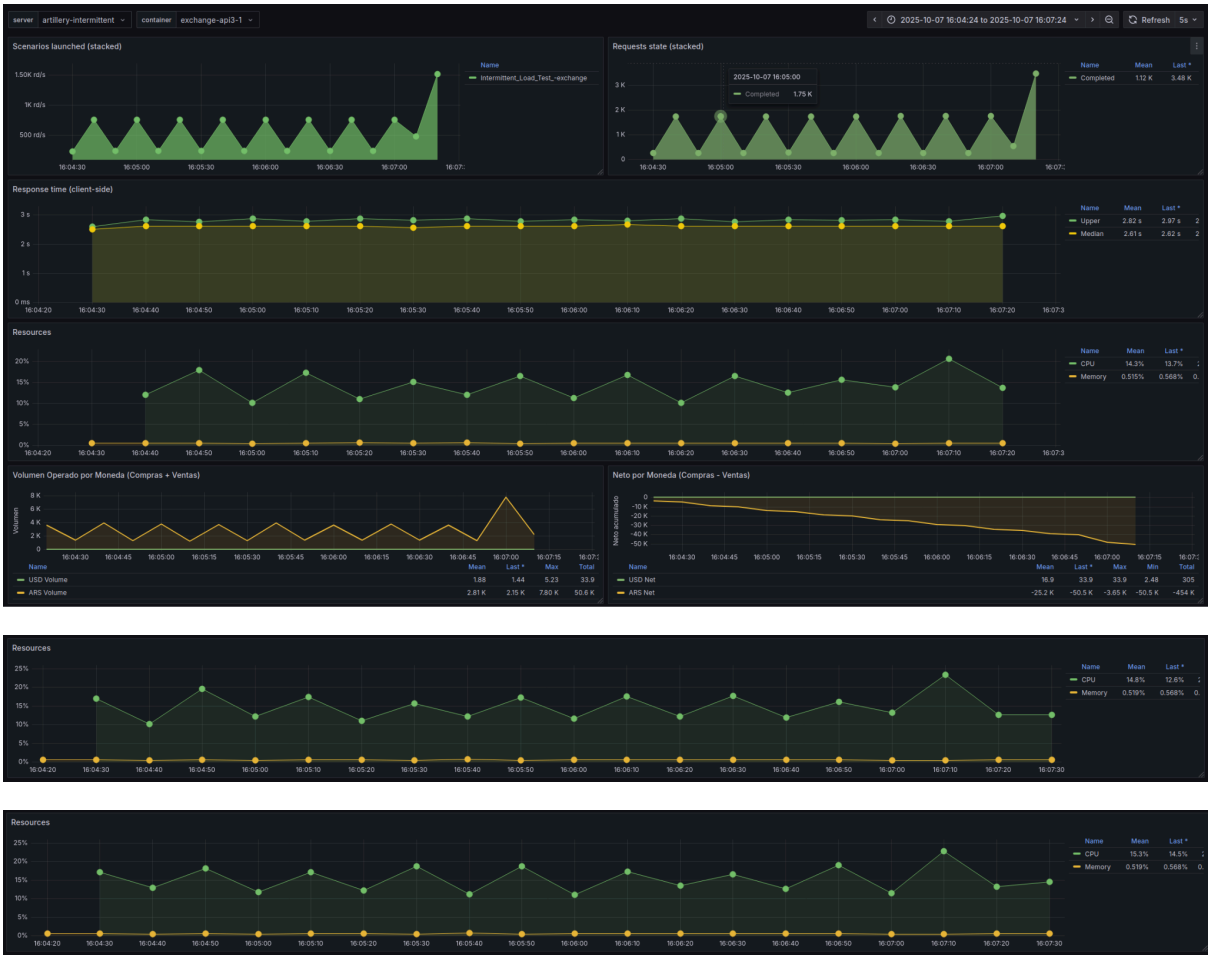
Abajo adjuntamos el uso de recursos de la segunda y tercera réplica.





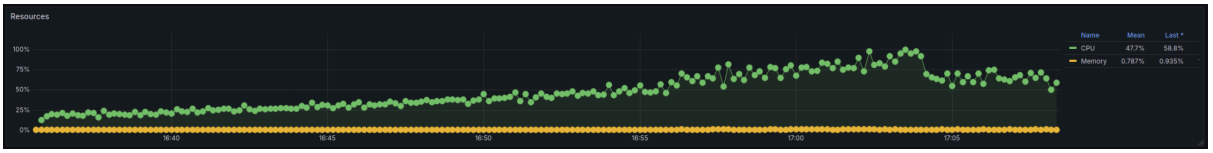
También cabe destacar que el uso de CPU se redujo como era esperable al agregar más réplicas, pero esto no significa que se están aprovechando adecuadamente. Esto nos informa de que las solicitudes no están fallando por falta de recursos en el procesamiento de estas, si no porque el no limitar su llegada adecuadamente provoca que la mayoría no lleguen ni a intentar procesarse. Estas solicitudes fallan con el código de error 499, que se refiere a un cierre de conexión inesperado por parte de los clientes.

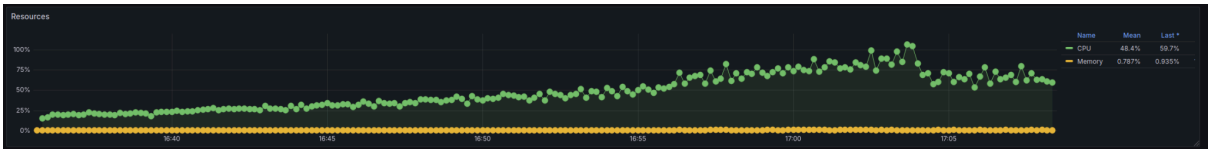
Caso 4: Intermitente (Replicación)



No hay mucho que remarcar en este gráfico, la única diferencia es la reducción del uso de recursos en cada nodo individual, las latencias se mantuvieron constantes en los picos y los volúmenes intercambiados y netos de monedas son acordes a los movimientos realizados.

Caso 5: Stress Test (Replicación)

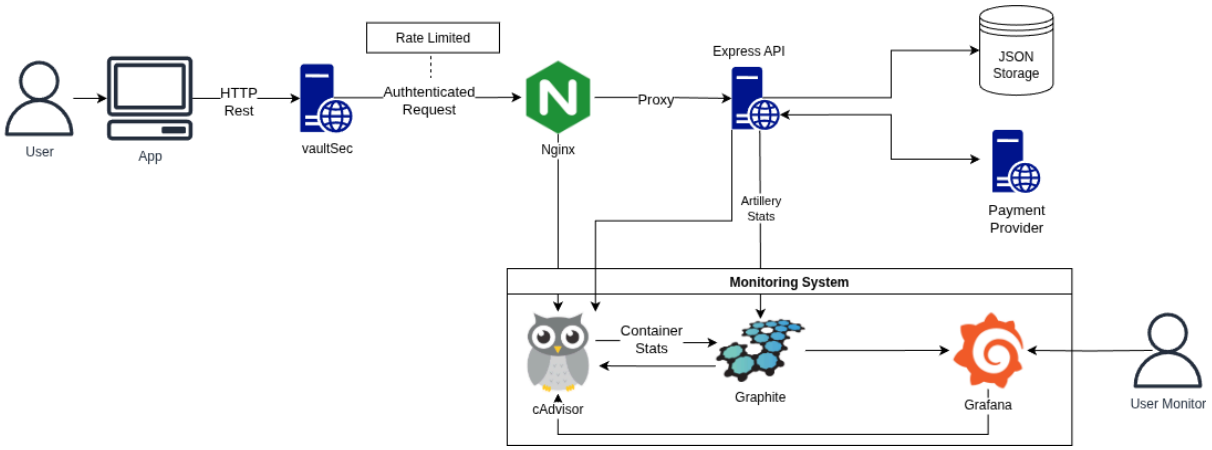




Para este caso nos tomamos la libertad de bajar los límites para los cuales el servidor empieza a responder con error para no tener escenarios muy caóticos cuando el resultado ya es predecible viendo las métricas para estos casos de replicación y los primeros. Los resultados se condicen con lo visto hasta ahora. no ganamos mucha mas disponibilidad de la que había pero vemos que en cuanto a recursos, cada réplica usa una fracción de los recursos del caso original

Entonces, como conclusión final, la replicación en nuestro caso no logró obtener resultados relevantes para poder resolver el caso principal de los usuarios que fue inestabilidad en la funcionalidad de intercambio de monedas, lo que sí se pudo ver fue una distribución de los recursos usados en las distintas instancias, lo que puede ser útil para escalar el sistema una vez que este tenga operaciones de cómputo intensivas (no siendo actualmente el caso)

Rate Limiting

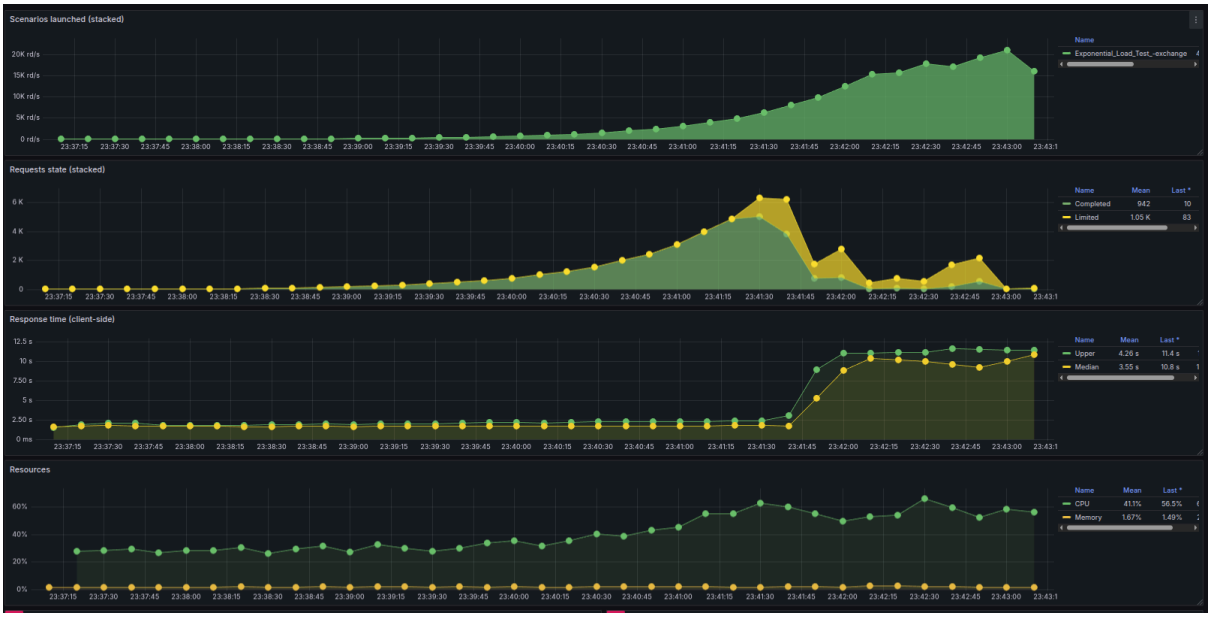


Dando más protagonismo al nodo Nginx, introducimos un **Rate Limiting** para filtrar la cantidad de requests simultáneas que llegan al servidor. Esta es una de las tácticas que hoy en día más se utilizan, dado que es un cambio ligero pero con muy buenos resultados. Para esto, hubo que cambiar ligeramente la configuración de nginx, introduciendo la keyword *limit_req* con algunos parámetros que podemos obtener a partir de las métricas anteriores. Para este caso, vamos a mantener un límite de 500 requests por segundo para darle tiempo a las instancias de procesar correctamente todos los pedidos, también vamos a configurar que los pedidos que lleguen muy seguido se encolen para que su llegada al servicio tenga una distribución más uniforme entre solicitudes y las requests que excedan este límite impuesto fallen con un código de error apropiado.

A continuación se verán las métricas probadas para este caso. Queremos hacer un par de aclaraciones:

- Los escenarios que **no** se probaron en este caso fueron el caso base de Rates y el caso de Stress, ya que consideramos que no aportan valor para este análisis en particular
- Redujimos la cantidad de requests en casos que fallaban abruptamente para que esté a niveles manejables para el Rate Limiter sin llegar a saturarlo completamente como pasó en otros casos anteriores.

Caso 1: Exponencial (Rate Limiting)

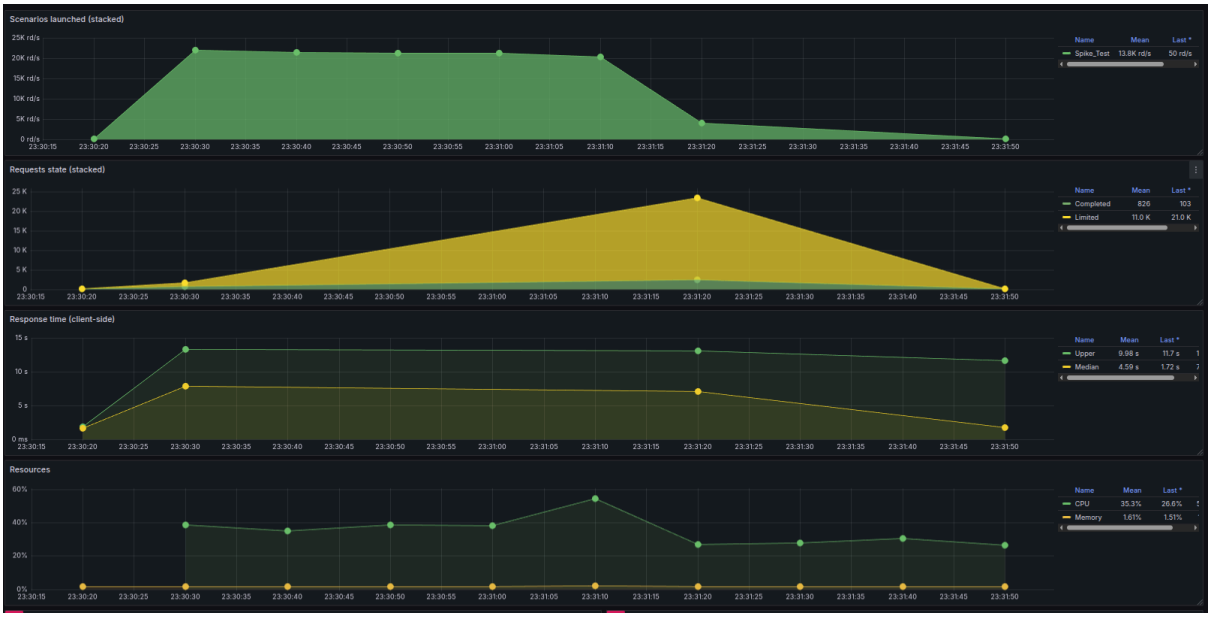


En esta prueba se busca analizar el comportamiento ante un crecimiento progresivo del tráfico cuando se aplica un rate limiter. A diferencia del escenario sin control, el crecimiento exponencial de la carga no provoca una saturación abrupta, ya que el rate limiter actúa restringiendo la cantidad de peticiones simultáneas cuando el sistema se acerca a su límite.

En los gráficos se observa que las solicitudes completadas (en verde) se mantienen constantes, mientras que las limitadas (en amarillo) aumentan a medida que crece la demanda. Esto demuestra que el sistema no se ve forzado a procesar más solicitudes de las que puede manejar, estabilizando la carga antes de alcanzar la saturación, mejorando la disponibilidad del servicio.

La latencia no tuvo grandes variaciones, en parte gracias al encolado de requests limitadas para su posterior procesamiento, demostrando una buena performance en comparación con a la prueba sin rate limiting. El uso de CPU se mantiene alrededor del 50 % durante la carga más densa, reflejando una mejoría sobre el caso base sin rate limiting el cual llegó a tocar valores de 150% (implicando el uso de más de 1 core).

Caso 2: Spike (Rate Limiting)



En este gráfico podemos ver el mismo patrón triangular de aumento y descenso rápido de carga, pero, a diferencia del test original, la cantidad de solicitudes limitadas (en amarillo) crece considerablemente durante el pico.

Esto indica que el limitador entra en acción inmediatamente, evitando que el servidor procese más operaciones de las que puede sostener y protegiendo la aplicación de caídas, evitando fallas en la disponibilidad del servicio. Como resultado, las solicitudes completadas, aunque sean menos que antes, se mantienen estables y la latencia, aunque presenta un leve aumento durante el pico, no se dispara de forma crítica.

El uso de CPU alcanza valores moderados cercanos al 60% y se reduce rápidamente al 30% tras el descenso de la carga, una notable mejoría con respecto al caso base.

Caso 3: Intermitente (Rate Limiting)

En este escenario se aprecia un patrón de subidas y bajadas regulares en la cantidad de escenarios lanzados, donde el rate limiter controla eficazmente la cantidad máxima de requests permitidas por ciclo.

Se observa que todas las solicitudes se completan correctamente y no aparecen errores significativos. La latencia se mantiene baja y estable a lo largo de los distintos picos. En cuanto a los recursos usados, los resultados están alineados con los del caso base. Cabe destacar que en este escenario a diferencia de los anteriores (caso base y replicación) se ajustó la cantidad de solicitudes aumentándolas para que el rate limiter actúe en los picos.

Como conclusión del análisis de las métricas para este caso, podemos decir que la introducción del rate limiting tuvo sus beneficios en cuanto a la disminución de errores y mejor espaciado de las requests a la hora de llegar al servidor, haciendo que el uso de recursos sea más eficiente, y pudimos ver más casos exitosos en contraste con la otra propuesta implementada (Replicación).

Tácticas para aplicar respecto a otros atributos de calidad

Ya que elegimos dos tácticas para aplicar al servicio para analizar su rendimiento, en esta sección discutimos aquellas mejoras que podría también ser aplicadas:

- **Almacenamiento:** La replicación de instancias si bien tiene mejoras en los tiempos de respuesta al haber más disponibilidad, trae consigo una desventaja que está íntimamente ligada con el diseño de la aplicación. Ahora mismo, el sistema está guardando en disco los logs, los saldos de las cuentas y los tipos de cambio. Esto se hace por servicio, entonces los distintos servicios tienen estados que no se comparten, haciendo que algunos queden con cuentas con saldos o tipos de cambio distintos, haciendo que la **consistencia** de los datos se vea afectada (en unos containers se ve una tasa de cambio que no tiene por que ser igual en los otros, dada la naturaleza del load balancer, y dependiendo de hacia donde vaya a ser procesada nuestra solicitud, lo que tendría sentido es que funcione igual en todos los servicios, es decir, ser **transparente** para el usuario). Presentamos dos posibles soluciones a este problema, con sus ventajas y desventajas:
    - **Volumen compartido:** Podemos hacer un cambio en la infraestructura de nuestro sistema, haciendo que los archivos queden centralizados en un volumen que sea accesible para todas las instancias del servicio. Con esto parecería que el problema de los estados inconsistentes queda solucionado, porque todas las instancias leen y escriben en un solo volumen, pero esto puede llegar a complejizar aún más el sistema, porque ahora nos tenemos que preocupar por race conditions que puedan aparecer, así como lecturas y escrituras “dirty”. Un ejemplo de este caso puede ser el caso donde una transferencia y una modificación en la tasa de cambio entre las monedas que están involucradas esa transferencia: dependiendo de cual se aplique primero, nuestro sistema puede quedar en un estado invalido (como por ejemplo saldo negativo). Así que lejos de solucionar este problema, la alternativa lo empeora agregando complejidad y un poco de latencia al tener el volumen separado.

    - **Base de datos:** El almacenamiento puede tercerizarse a una Base de Datos que maneje todas las solicitudes por nosotros. Esto tiene la ventaja de centralizar el acceso a los datos y no preocuparnos por los problemas de concurrencia o inconsistencia que si teníamos con el volumen compartido, como desventaja podemos nombrar el overhead adicional que tiene la base de datos, y un poco de delay en la comunicación (ya que hay que establecer una conexión con esta base, no es tan sencillo como guardar a disco), y el almacenado en un solo nodo lo vuelve un Single Point of Failure. Para este caso implementamos un almacenamiento en Redis (para una rama en el repositorio), donde primero aplicamos una migración de datos si hacen falta desde los archivos JSON iniciales y luego la aplicación se comunica con redis directamente, dejando deprecada la forma de almacenamiento inicial.
- **Tests:** Es posible aplicar tácticas en cuanto a la Testeabilidad del sistema, como por ejemplo la implementación de tests unitarios, tests de integración y tests End to End, y si queremos favorecer aún más a la Disponibilidad del sistema, podemos ser aún más robustos y generar pipelines de Integración Continua (CI/CD) o Canary Deployments para mantener un nivel alto de robustez, y nos aseguramos de minimizar los problemas cuando este proyecto esté en producción.
- **Validación de los datos que ingresan al sistema:** Si bien se menciona que la interfaz de usuario es la que maneja los formatos y los montos del usuario, estamos a una request con datos inválidos de hacer que nuestro sistema deje de responder. Un cambio poco costoso pero eficiente es manejar los estados que hacen que la aplicación falle antes de que sea un problema para nosotros, por ejemplo, si los tipos de monedas en una transferencia no incluyen “ARS”, poder fallar correctamente informando al usuario y manteniendo vivo al servicio.
- **Monitoreo de los servicios:** Podemos implementar sistemas de Health-check o Heartbeats para los servicios principales, para poder reiniciarlos en caso de alguna caída que deje al servicio sin responder. Si se quiere implementar un sistema verdaderamente disponible, esto es un requerimiento que no se nos tiene que pasar por alto.