Informe Trabajo Práctico Nro 1

Facultad de Ingeniería
Universidad de Buenos Aires

Arquitectura del Software

Facultad de Ingeniería, Universidad de Buenos Aires

Abril, 2025

Alumnos:

Nombre	Padrón
Iván Grzegorczyk	104084
Caffaratti, Maria Cecilia	72629
Juan Manuel Diaz	108183
Gabriel Morbello	108552

Tabla de contenidos

Informe

1. Introducción y contexto del problema

En el presente informe se detalla el proceso de análisis, optimización y medición de métricas llevado a cabo en el proyecto **arVault**, un sistema basado en arquitectura de microservicios con tecnologías como Node.js, Nginx y Docker.

El objetivo principal fue implementar un sistema de monitoreo que permitiera registrar el volumen operado y el neto de transacciones, además de optimizar el almacenamiento de datos para mejorar la eficiencia general del sistema.

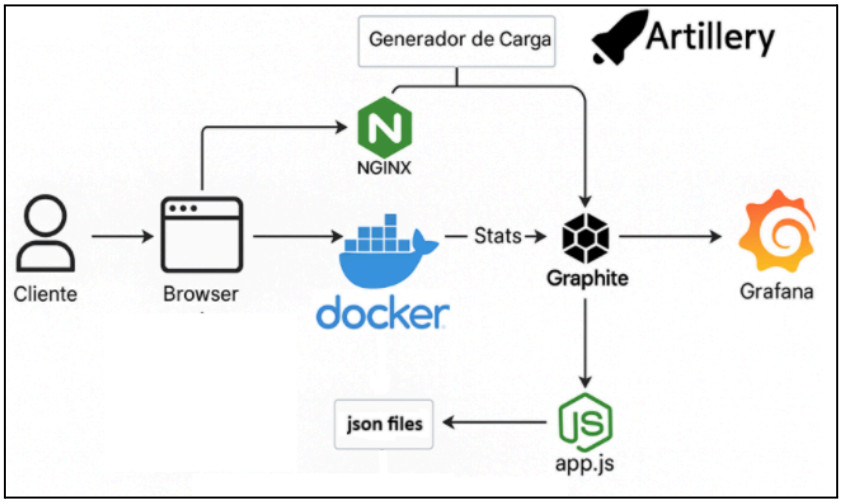
Se trabajó en la configuración de herramientas de monitoreo (Artillery, StatsD, Graphite, Grafana) y en la implementación de Redis como sistema de almacenamiento en memoria. Posteriormente, se realizaron pruebas de carga para medir el impacto de las mejoras implementadas.

2. Análisis del Caso Base ("estado inicial")

Diagrama Components & Connectors (C&C) - Estado Base

Vista General

El sistema está compuesto por varios servicios que interactúan entre sí para ofrecer una aplicación web robusta, utilizando métricas para el monitoreo del rendimiento. Todos los componentes están orquestados a través de **Docker** y definidos en un archivo `docker-compose.yml`, lo que garantiza portabilidad, facilidad de despliegue y consistencia del entorno.



1. Usuario (Browser)

- **Función:**
  Los usuarios interactúan con la aplicación web mediante un navegador enviando solicitudes HTTP.

- **Importancia:**
  Es el punto de entrada del flujo de solicitudes.

2. Nginx (Proxy Reverso)

- **Función:**
  Nginx recibe las solicitudes de los usuarios y las reenvía de manera eficiente al backend de Node.js.

- **Importancia:**
  Permite balancear carga, manejar HTTPS y optimizar el tráfico, mejorando el rendimiento y la seguridad.

3. Backend Node.js (Express)

- **Función:**
  Servicio de backend que maneja la lógica de negocio, procesa solicitudes, realiza operaciones de cambio de divisas y registra métricas del sistema.

- **Importancia:**
  Es el núcleo de la aplicación; todo el procesamiento de datos ocurre aquí.

4. StatsD (Cliente de Métricas)

- **Función:**
  Recibe métricas generadas por Node.js (como volumen de operaciones y montos netos) y las reenvía en tiempo real hacia Graphite.

- **Importancia:**
  Facilita la recopilación de métricas de manera ligera y eficiente.

5. Graphite (Almacenamiento de Métricas)

- **Función:**
  Base de datos de series temporales que almacena las métricas recolectadas.

- **Importancia:**
  Permite la persistencia y consulta de métricas históricas para análisis de rendimiento.

6. Grafana (Visualización de Métricas)

   - **Función:**
     Plataforma de visualización que consulta Graphite para construir dashboards interactivos y visuales.

   - **Importancia:**
     Proporciona a los desarrolladores y administradores una vista clara del estado del sistema mediante gráficos y alertas.

7. Docker

   - **Función:**
     Todos los componentes (Nginx, Node.js, StatsD, Graphite, Grafana) están dockerizados, lo que significa que corren en contenedores aislados.

   - **Importancia:**
     Docker garantiza que la aplicación sea portable, escalable y fácil de desplegar en cualquier entorno, eliminando problemas de configuración y dependencias.

8. Artillery (Generador de Carga)

   - **Función:**
     Herramienta de pruebas de rendimiento que genera tráfico simulado hacia el backend para medir su comportamiento bajo diferentes volúmenes de carga.

   - **Importancia:**
     Permite evaluar el rendimiento, la escalabilidad y detectar cuellos de botella antes de la puesta en producción.

Despliegue de Servicios

El despliegue de todos los servicios se realiza utilizando **Docker Compose**. Cada servicio tiene su propio contenedor, con las siguientes consideraciones:

- **Nginx:** Expone el puerto 5555 y reenvía solicitudes al backend.

- **Node.js:** Corre sobre el puerto 4000 y maneja todas las operaciones de negocio.

- **StatsD:** Escucha métricas en UDP y las reenvía a Graphite.

- **Graphite:** Se encarga de almacenar métricas, expone la interfaz web en el puerto 8080.

- **Grafana:** Disponible en el puerto 3000, permite la visualización gráfica de las métricas.

- **Artillery:** Se ejecuta localmente para simular tráfico y generar métricas durante las pruebas de carga.

Flujo de Solicitudes y Datos

El flujo general de interacción y transmisión de datos en el sistema es el siguiente:

1. **El usuario** accede a la aplicación web y realiza una solicitud HTTP (por ejemplo, para cambiar moneda).

2. **Nginx**, actuando como proxy reverso, recibe esta solicitud y la redirige al servicio backend de **Node.js**.

3. **Node.js (Backend)** procesa la solicitud:

    - Valida los datos de entrada.

    - Calcula el resultado de la operación de cambio de divisa.

    - Actualiza internamente el volumen total operado y el neto de transacciones.

4. **Node.js** envía métricas de operación (como volumen y neto) hacia **StatsD**.

5. **StatsD** recibe estas métricas y las retransmite en tiempo real a **Graphite**.

6. **Graphite** almacena las métricas como series temporales.

7. **Grafana** consulta los datos de **Graphite** y actualiza los dashboards, permitiendo una visualización clara y detallada del estado actual y pasado del sistema.

8. **Artillery** puede ser ejecutado para generar tráfico simulado, aumentando la cantidad de solicitudes de usuario y así testear el comportamiento de la infraestructura bajo carga.

Análisis de QA (Quality Attributes)

En esta tabla podemos ver en resumen el análisis de los principales indicadores de calidad

QA (atributo de calidad)	Evaluación	Comentarios
Availability	🔴 Baja	Si se cae el proceso o se corrompe un archivo `.json`, el sistema falla. No hay redundancia ni tolerancia a fallos. No existe sistema de reinicio automático del servicio. El sistema es simple, sencillo de levantar y rápido para recuperar de un backup.
Scalability	🔴 Baja	El uso de archivos JSON impide el escalamiento horizontal para manejar múltiples requests en paralelo ya que no hay control de acceso concurrente a los archivos.
Maintainability	🟡 Regular	El código mezcla lógica de negocio con persistencia. El endpoint `/accounts/:id/balance` modifica saldos directamente.
Performance	🟡 Regular	Lectura/escritura a disco cada 5s. Escala mal. No hay caché. Los archivos pueden volverse lentos al crecer. Realizar todas las operaciones en memoria lo hace muy performante, a costa de no poder escalar.
Modifiability	🟢 Buena	Está todo en un solo lugar y sin validaciones complejas. Fácil de cambiar, pero poco robusto.
Observability	🔴 Nula	No hay métricas, logs del sistema o visibilidad del estado operativo. Tampoco hay alertas.
Security	🔴 Baja	El manejo incorrecto de requests concurrentes atenta contra la integridad de los datos, llevando a que se realicen transacciones con saldo insuficiente. La manera en que se guardan los archivos lleva a que se pueda perder toda la información; backups insuficientes (más detalles en Reliability).
Reliability	🔴 Baja	Como los archivos JSON se abren en modo escritura, una falla durante la escritura causaría que los archivos estén vacíos, eliminando el único backup. Fallas en cualquier componente (Express, NGINX) implicarían una falla total del sistema. No hay redundancia a fallos.

Se evaluaron los principales atributos de calidad del sistema actual.

A continuación, se detallan los hallazgos:

Availability

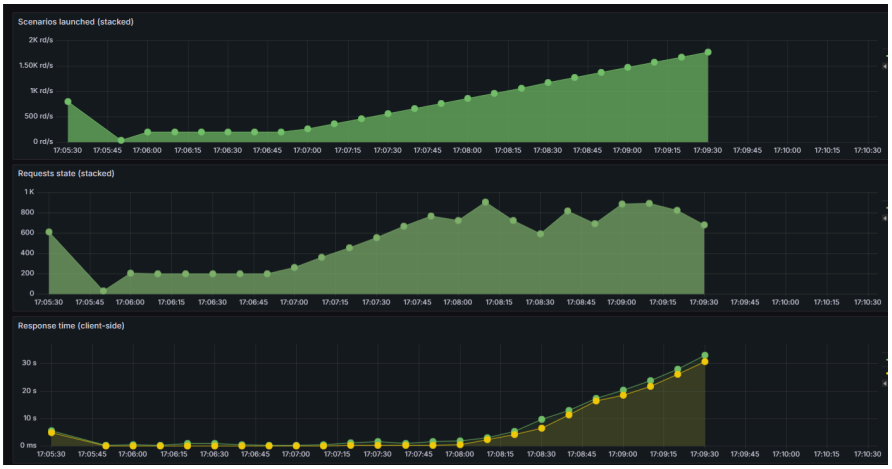
- **Evaluación:** 🔴 Baja
- **Comentarios:** El sistema depende del correcto funcionamiento del servicio de Node.js y de la integridad de los archivos `.json`. No existen mecanismos de redundancia, failover o tolerancia a fallos. Ante una caída del proceso o corrupción de datos, el servicio deja de funcionar. Tampoco se garantiza el reinicio de la app ante una caída.

  En caso de tener que recuperarse de un backup, el RTO es bajo considerando que el sistema solo tiene que levantar NGINX y leer pocos archivos JSON, y el RPO también es bajo, siendo el último backup como máximo 5 segundos antes de la falla.

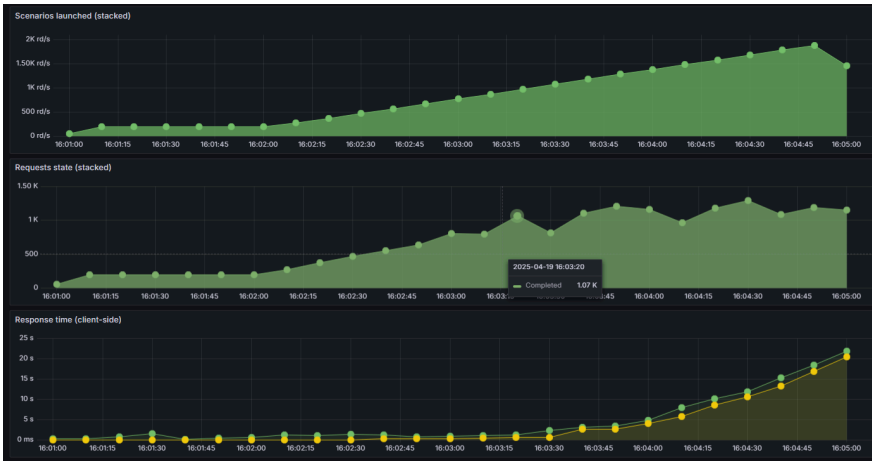
Scalability

- **Evaluación:** 🔴 Baja
- **Comentarios:** El diseño actual basado en almacenamiento de archivos `.json` presenta limitaciones severas frente a múltiples accesos concurrentes. No hay control de concurrencia ni bloqueo de acceso a los archivos, lo cual genera cuellos de botella a medida que crece la cantidad de usuarios o transacciones. Si bien se puede escalar verticalmente, y esto conlleva una mejora en la capacidad de respuesta (en cantidad de requests que puede responder por unidad de tiempo la api), este escalado en particular requiere la interrupción del servicio, sin embargo, el escalado vertical tendría también un límite superior (que bajaría la efectividad de escalado), dado que hay un tope de requests que se pueden resolver en simultáneo, sin entrar en problemas de concurrencia en el grabado de los archivos. Por otro lado, no sería posible escalar horizontalmente (con la arquitectura inicial, sin modificaciones necesarias), ya que tanto las transacciones, como los tipos de cambios establecidos (datos en el `state`), quedarían **desincronizados** entre las instancias de la app, y esto no permitiría el correcto uso de la misma (el objetivo de la app es poder mantener un mismo estado, y operar sobre eso).

  Realizamos un pequeño análisis de escalabilidad vertical de la arquitectura actual, donde se puede evidenciar una mejora de rendimiento (que igualmente, y como ya aclaramos, está limitada por la arquitectura).



En este primer caso podemos observar una variación de más de 10 segundos en el tiempo de respuesta de la api, en el endpoint de rates, al alcanzar las 120 requests por segundo.



En el segundo caso, habiendo mejorado el contenedor para tener un 20% más de capacidad de cpu, se observa esta variación recién cerca de las 150 requests por segundo.

Maintainability

- **Evaluación:** 🟡 Regular
- **Comentarios:** El sistema presenta una mezcla entre la lógica de negocio y la persistencia de datos, dificultando la separación de responsabilidades. Por ejemplo, el endpoint `/accounts/:id/balance` altera directamente los saldos, lo que complica futuras modificaciones o extensiones de funcionalidad.

Performance

- **Evaluación:** 🟡 Regular
- **Comentarios:** La lectura y escritura constantes sobre disco, sin mecanismos de cacheo, impacta negativamente en el rendimiento. A medida que los archivos `.json` crecen en tamaño, las operaciones de entrada/salida (I/O) se vuelven más costosas, degradando el tiempo de respuesta.

Modifiability

- **Evaluación:** 🟢 Buena
- **Comentarios:** Debido a la simplicidad del diseño y la baja complejidad de las validaciones, realizar cambios en la funcionalidad es relativamente sencillo. Sin embargo, esta facilidad implica una falta de robustez en el manejo de errores y casos de borde.

Observability

- **Evaluación:** 🔴 Nula
- **Comentarios:** El sistema carece de mecanismos de monitoreo. No se recolectan métricas, logs operativos ni alertas ante fallos o anomalías, lo cual impide realizar un diagnóstico proactivo de problemas.

Security

- **Evaluación:** 🔴 Baja
- **Comentarios:** La seguridad en términos de interacción con clientes externos no es gestionada por el sistema actual, sino que es responsabilidad de un componente externo (vaultSec), quedando fuera del alcance de este análisis.
  En cuanto a la seguridad de la aplicación misma, existe una vulnerabilidad que atenta contra la integridad de los datos. Al recibir múltiples peticiones concurrentes para transacciones, el sistema intenta manejarlas de manera concurrente dada la naturaleza del código async. Esto lleva a que todas las transacciones validen que la cuenta tenga saldo suficiente, y después transfieran ese saldo a otra cuenta, incluso cuando la cuenta no tenía saldo suficiente para *todas* las transacciones.

Reliability

- **Evaluación:** 🔴 Bajo
- **Comentarios:** El sistema presenta problemas al almacenar los datos que persisten en memoria, lo que puede provocar errores en el procesamiento de los files frente a la interrupción de los jsons. En caso de no poder guardar correctamente los archivos, por un lado, no podría detectarse este error, y por otro, al querer levantar nuevamente el servicio, habríamos perdido toda la información de las transacciones realizadas y modificaciones a los tipos de cambio.
  Además, los backups podrían estar corruptos y ser posteriores a la falla, lo correcto sería también persistir versiones históricas del backup durante un cierto período de tiempo.

Análisis Crítico del Diseño Actual

El sistema actual presenta un diseño extremadamente simple, basado en una arquitectura monolítica sin capas de abstracción claras y con persistencia de datos en archivos `.json`.
Si bien esto puede ser adecuado para prototipos o cargas muy bajas, surgen varias limitaciones críticas:

- **Persistencia frágil:** El uso de archivos locales como única fuente de datos expone al sistema a corrupción y pérdida de datos. El sistema es funcional pero frágil. Está bien como MVP o demo, pero no para producción real. La lógica de negocio no está separada de la infraestructura (por ejemplo: el manejo de archivos ocurre dentro del mismo código que recibe el request).
- **Baja tolerancia a fallos:** No existe redundancia ni recuperación automática en caso de error. Los endpoints no tienen validaciones serias: cualquier input válido en forma se procesa sin chequeo real.

- **Problemas de concurrencia:** Accesos simultáneos a archivos `.json` pueden generar corrupción de datos.
- **Escalabilidad limitada:** El sistema no está preparado para manejar múltiples usuarios concurrentes ni un crecimiento de volumen. El logeo cada 5 segundos es una solución "manual" poco eficiente. Podría haber logs en tiempo real o buffers más inteligentes. El almacenamiento en archivos JSON no garantiza ni velocidad ni escalabilidad para un entorno de alta concurrencia.
- **Observabilidad inexistente**: No se pueden medir ni anticipar fallos ya que no hay monitoreo del estado de la aplicación. Ausencia de métricas de rendimiento, lo que dificulta el análisis de la eficiencia del sistema.
- **Performance dependiente del tamaño:** A medida que los archivos crecen, el sistema se vuelve cada vez más lento, afectando la experiencia del usuario. Potenciales cuellos de botella en Nginx debido a la configuración por defecto.
- **Mantenibilidad aceptable pero riesgosa:** Aunque es fácil de modificar, la falta de separación de responsabilidades aumenta el riesgo de introducir errores accidentales.

El diseño actual es funcional para un entorno controlado de baja carga, pero no cumple con los estándares mínimos de sistemas robustos o de producción.

3. Tácticas aplicables para mejorar

QA	Táctica de arquitectura	Posible implementación
Performance	Caché, Batch, reducir I/O	Caché en memoria o Redis. Persistencia async.
Availability	Redundancia, retry, timeouts	Contener errores, circuit breaker si crece.
Scalability	Stateless, external DB, load balancer	Desacoplar storage, usar Redis/PostgreSQL, balanceo con nginx.
Modifiability	Separación de responsabilidades	Separar el módulo de cuentas y el de exchange.

Observability	Logs, métricas, tracing	Agregar metricas- StatsD + Grafana. Agregar logs estructurados.
Reliability	Transacciones, persistencia segura	Usar Redis o base con atomicidad. Validaciones.
Security	Transacciones, locking	Si se usa Redis, aprovechar la atomicidad.

1- Implementación de Redis como Motor de Persistencia:

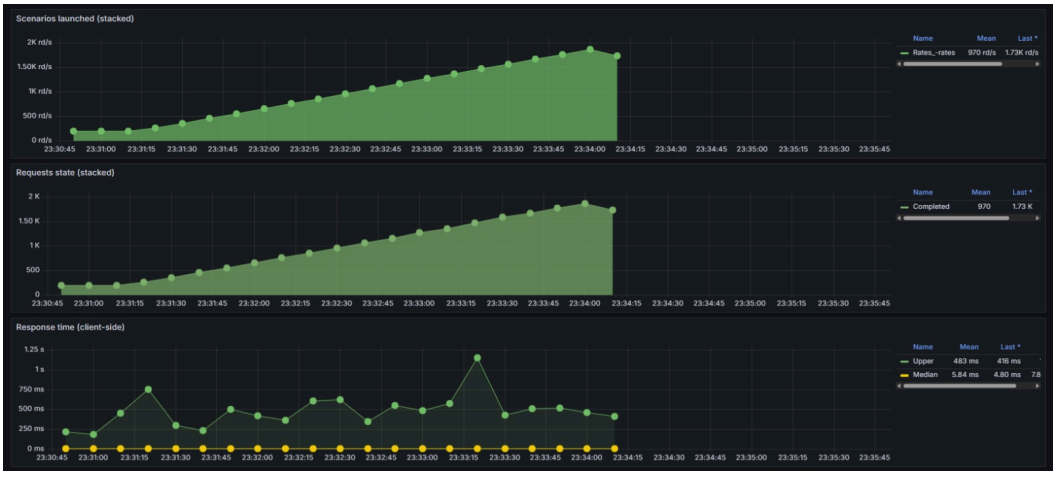
- **Availability:** Mejora. Redis es más robusto que archivos locales y permite persistencia en memoria con mecanismos de replicación (en una futura mejora).
    - *Antes:* Si el archivo `.json` se corrompe, el sistema queda inutilizado hasta que alguien intervenga manualmente.
    - *Después:* Redis, aunque en esta implementación básica no tiene replicación, ofrece una estructura de persistencia más confiable, con recuperación en memoria y manejo de concurrencia. Se reduce el riesgo de corrupción de datos. Existe el riesgo de que se caiga el redis dejando todo el sistema sin funcionamiento. Podrían implementar mecanismos para detectar esta falla e intentar levantarlo nuevamente.
- **Scalability:** Mejora significativa. Redis soporta múltiples conexiones concurrentes y operaciones atómicas de lectura/escritura.
    - *Antes:* Solo un proceso puede acceder de manera segura al archivo `.json` al mismo tiempo. Cada contenedor tiene una copia distinta de los archivos lo cual anula la posibilidad de escalar de forma horizontal
    - *Después:* Redis maneja concurrencia nativamente y permite miles de operaciones por segundo sin bloqueos de lectura/escritura. Permite escalar horizontalmente debido a que los datos quedan centralizados en un servicio distinto al de la api.
- **Performance:** Mejora importante. Redis opera en memoria, reduciendo latencias de acceso de milisegundos a microsegundos.
    - *Antes:* Cada operación requería abrir, modificar y guardar un archivo de disco, un proceso lento y costoso en I/O.
    - *Después:* Redis guarda los datos en memoria RAM, proporcionando acceso ultrarrápido a la información.
- **Maintainability:** Ligera mejora. La separación de la lógica de negocio respecto de la persistencia se vuelve más clara usando un servicio dedicado.
    - *Antes:* La lógica de negocio debía entender cómo manipular archivos locales, mezclando responsabilidades.
    - *Después:* Separación clara: el almacenamiento pasa a ser responsabilidad de Redis, y la app de Node.js sólo interactúa mediante el cliente de Redis.

Para la incorporación de Redis a la solución, seguimos lo siguientes pasos:

1. Se modifica el `docker-compose.yml` agregando el servicio redis con persistencia en un volúmen
2. Se instala el cliente Redis para Node.js
3. Se agrega un módulo para conectarse al mismo al iniciar la api
4. Se modifica el archivo de `state` para almacenar y obtener los datos a través del cliente de redis.

Con los datos centralizados en el servicio de redis pudimos hacer una prueba de escalamiento horizontal obteniendo los siguientes resultados (escenario rates.yaml) con una sola instancia:

Al escalar la api a 4 instancias y distribuir la carga con nginx obtuvimos los siguientes resultados:



En base a esto podemos concluir que escalar horizontalmente la api, usando Nginx como balanceador de carga y redis como base de datos persistente, logra un incremento significativo en la capacidad de manejo de carga y una notable mejora en la consistencia de los tiempos de respuesta. El sistema es capaz de completar más solicitudes por segundo sin bajar el rendimiento, manteniendo baja latencia y estabilidad.

2- Instrumentación de Métricas (StatsD, Graphite, Grafana):

- **Observability:** Mejora radical. Permite visualizar en tiempo real el estado del sistema (volumen operado, tiempos de respuesta, errores, etc.).
    - *Antes:* No había forma de saber cuántas operaciones se hacían, el estado de salud del sistema, o si había fallos.
    - *Después:* Se recolectan métricas de negocio (volumen y neto de transacciones) y métricas operativas (latencia, tasa de errores), que pueden visualizarse en tiempo real en Grafana.

- **Performance:** Mejora indirecta. Identificar cuellos de botella permite optimizar el sistema.
    - *Antes:* No había forma de identificar si el sistema se volvía lento hasta que los usuarios se quejaran.
    - *Después:* Podemos monitorear tiempos de respuesta y detectar problemas antes de que impacten al usuario.

- **Availability:** Mejora indirecta. Permite detectar caídas tempranamente y alertar.
    - *Antes:* Si el sistema caía, nadie se enteraba hasta que era demasiado tarde.
    - *Después:* A futuro, los dashboards permiten configurar alertas automáticas para incidentes críticos (no en este TP pero sí en una implementación extendida).

Métricas incorporadas

1. Conteo de Solicitudes API

- **Métrica:** `api.requests`
- **Tipo:** `counter`
- **Descripción:**Cada vez que un endpoint de la API es invocado (GET/PUT/POST), se incrementa este contador.Permite conocer el volumen total de tráfico que maneja el backend.
- **Implementación:**

```
statsd.increment('api.requests'); // Incrementa el contador de solicitudes
```

2. Volumen de Operaciones de Exchange por Moneda

- **Métricas:**
    - `arvault.exchange.volume.<baseCurrency>`
    - `arvault.exchange.volume.<counterCurrency>`
- **Tipo:** `counter`
- **Descripción:**Mide la cantidad total de dinero operado en cada tipo de moneda, en las transacciones de cambio de divisas.Se actualiza tanto para la moneda base como para la moneda de contrapartida.

- **Implementación:**

```
// Enviar métricas a Graphite
statsd.increment(`arvault.exchange.volume.${baseCurrency}`, baseAmount);  // Incrementa volumen moneda base
statsd.increment(`arvault.exchange.volume.${counterCurrency}`, exchangeRequest.counterAmount); // Incrementa volumen moneda contraria
```

3. Neto de Transacciones por Moneda

- **Métricas:**
  - `arvault.exchange.neto.<baseCurrency>`
    `arvault.exchange.neto.<counterCurrency>`
- **Tipo:** `counter` (increment y decrement)
- **Descripción:** Representa el flujo neto de operaciones de compra/venta para cada moneda.
- **Implementación:**

```
statsd.increment(`arvault.exchange.neto.${baseCurrency}`, baseAmount);
statsd.decrement(`arvault.exchange.neto.${counterCurrency}`, exchangeRequest.counterAmount);
```

4. Tiempo de Respuesta del Endpoint `/exchange`

- **Métrica:** `api.exchange.latency`
- **Tipo:** `timing` (medición de milisegundos)
- **Descripción:**Mide el tiempo que tarda en completarse una operación de cambio de divisas, desde que el usuario envía el request hasta que recibe la respuesta.
- **Implementación:**

```
const start = Date.now();  // Empieza a medir el tiempo de la solicitud
```

. . .

```
const latency = Date.now() - start;  // Calcular el tiempo de respuesta
statsd.timing('api.exchange.latency', latency);  // Enviar la latencia de la operación
```

<u>Beneficios de las Métricas Implementadas</u>

- Permiten **monitorear en tiempo real** el volumen de uso y el rendimiento de la aplicación.
- Facilitan la **detección temprana de problemas** de latencia o caídas de servicio.
- Mejoran la **capacidad de análisis de negocio** al mostrar el flujo y volumen neto de operaciones por moneda.
- Son la base para establecer **alertas automáticas** ante comportamientos anómalos en producción.

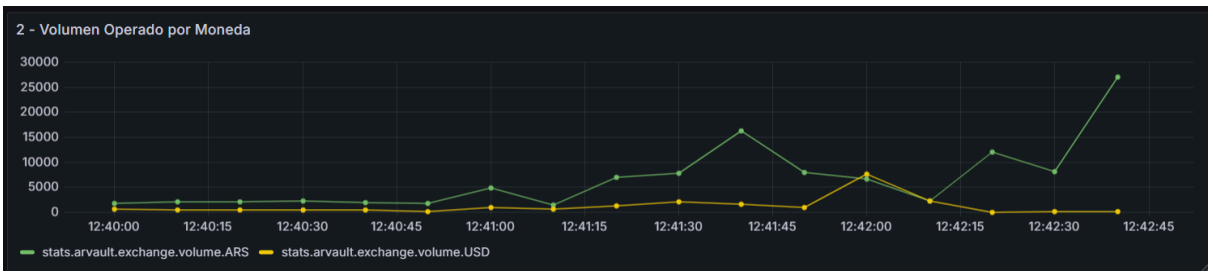
Diseño de Dashboards en Grafana

🎯Panel 1 - Total de Requests

- **Métrica:** `api.requests`
- **Tipo de gráfico:** Línea
- **Eje X:** Tiempo
- **Eje Y:** Número de solicitudes
- **Objetivo:**Monitorear la cantidad de requests procesadas por unidad de tiempo. Ayuda a entender la carga general de la API.

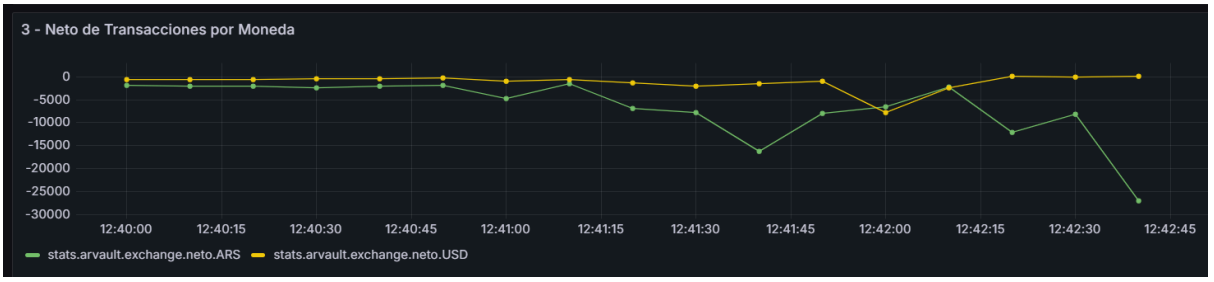
🎯 Panel 2 - Volumen Operado por Moneda

- **Métricas:** `arvault.exchange.volume.*`
- **Tipo de gráfico:** Área apilada
- **Eje X:** Tiempo
- **Eje Y:** Monto operado
- **Series:** Cada moneda (`USD`, `ARS`, `EUR`, etc.)
- **Objetivo:**Visualizar cómo evoluciona el volumen de operaciones para cada moneda.



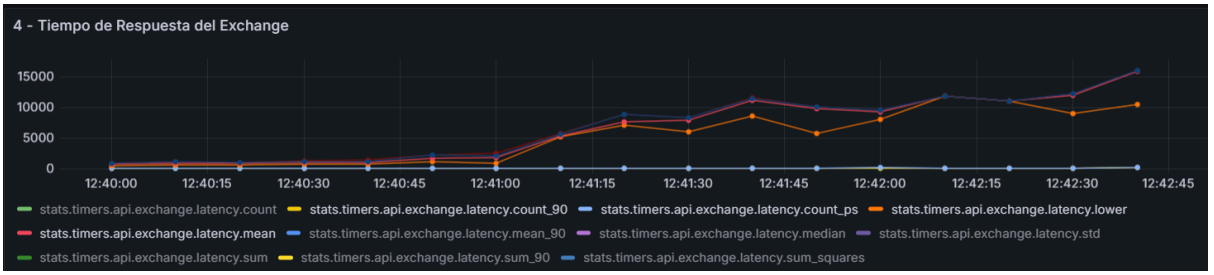
🎯 Panel 3 - Neto de Transacciones por Moneda

- **Métricas:** `arvault.exchange.neto.*`
- **Tipo de gráfico:** Línea
- **Eje X:** Tiempo
- **Eje Y:** Neto acumulado
- **Series:** Cada moneda
- **Objetivo:**Monitorear el flujo neto de compra/venta para cada tipo de moneda.



🎯 Panel 4 - Tiempo de Respuesta del Exchange

- **Métrica:** `api.exchange.latency`
- **Tipo de gráfico:** Box Plot o Línea
- **Eje X:** Tiempo
- **Eje Y:** Latencia en milisegundos
- **Objetivo:**Evaluar la performance de la operación más crítica (exchange). Detectar cuellos de botella o degradaciones de servicio.



Generación de Datos de Prueba

Para alimentar los dashboards con datos reales, se puede usar **Artillery** como generador de carga.

- Instalación de Artillery
- Crear los archivos de escenario (perf/scenario.yaml)
- Ejecutar el escenario (npm run artillery -- run escenario.yaml -e api)
- Validar que se envió tráfico a la aplicación y comenzará a recolectar métricas.

Los escenarios propuestos son:

- rates.yaml: Para validar la efectividad de las modificaciones en cuanto a la carga de trabajo (cantidad de requests que puede soportar la api), utilizamos, contar el endpoint /rates, una carga plana durante un minuto para establecer un flujo “constante” de carga, y seguido realizamos una rampa de requests para poder evidenciar la capacidad del servicio en las condiciones dadas.
- scenario7.yaml: Carga realista con errores, para realizar pruebas en el endpoint de /exchange
- scenario8.yaml: Prueba de estres, con más casos de exchange

- **Performance:** Mejora. Permite evaluar límites del sistema antes de una puesta en producción.
    - *Antes:* No había validación real del rendimiento del sistema bajo carga. Se asume que "funciona".
    - *Después:* Mediante Artillery podemos simular cientos de usuarios concurrentes y detectar hasta dónde el sistema aguanta sin degradarse.

- **Maintainability:** Mejora indirecta. Conocer los límites ayuda a planificar la escalabilidad futura de manera estructurada.
    - *Antes:* No había métricas objetivas para justificar mejoras o cambios.
    - *Después:* Tener datos duros sobre el comportamiento bajo carga permite documentar mejor la evolución del sistema, y justifica decisiones de arquitectura futuras.

3- Optimización de Nginx:

- **Performance:** Mejora. Una mejor configuración de Nginx permite un manejo más eficiente de conexiones concurrentes.
    - *Antes:* Nginx tenía su configuración por defecto, posiblemente con bajo número de procesos y conexiones permitidas.
    - *Después:* Con `worker_processes auto` y `worker_connections 1024`, Nginx puede aprovechar todos los núcleos disponibles y manejar más conexiones simultáneamente, mejorando el throughput.
- **Availability:** Mejora. Al tener un proxy eficiente, se reduce la probabilidad de saturación de recursos.
    - *Antes:* Un número bajo de conexiones simultáneas podía hacer que usuarios no pudieran conectarse si el sistema tenía carga.
    - *Después:* Aumentar la capacidad de Nginx mejora la disponibilidad al permitir que más usuarios se conecten al mismo tiempo sin bloqueos.

En resumen

Mejora	QA Mejorado	Impacto específico
Uso de Redis	Disponibilidad, Performance, Escalabilidad, Mantenibilidad	Persistencia segura y rápida.
Instrumentación de Métricas (StatsD + Graphite + Grafana)	Observabilidad, Disponibilidad, Performance	Visibilidad en tiempo real del estado del sistema.
Optimización de Nginx	Performance, Disponibilidad	Mejora del manejo de concurrencia.
Uso de Pruebas de Carga (Artillery)	Performance, Mantenibilidad	Análisis preventivo de cuellos de botella.

Diagrama de Componentes con las mejoras