Estrategias de observabilidad y tolerancia a fallos.

—

Alumnos: Bailon Mauro, Sandoval Federico.
Profesores: Marcia Tejeda y Mariano Claveria.
Arquitectura de Software II.
Universidad Nacional de Quilmes.

Contexto

El objetivo de este trabajo fue desarrollar y documentar estrategias de tolerancia a fallos y observabilidad bajo un enunciado donde se tuvo que consumir una api externa para obtener diferentes climas (https://openweathermap.org/).

En el presente informe se detallan las decisiones tomadas, tecnologías utilizadas, diferentes diagramas de secuencia, el diagrama de arquitectura, conclusiones sobre tests de carga y estrategias de alerting.

Casos de uso

Caso de uso	CU-1
Título	Obtener clima actual
Actor	Cliente
Descripción	Como cliente quiero consultar el clima actual.

Caso de uso	CU-2
Título	Obtener climas del ultimo dia
Actor	Cliente
Descripción	Como cliente quiero obtener todos los climas registrados en las ultimas 24hs.

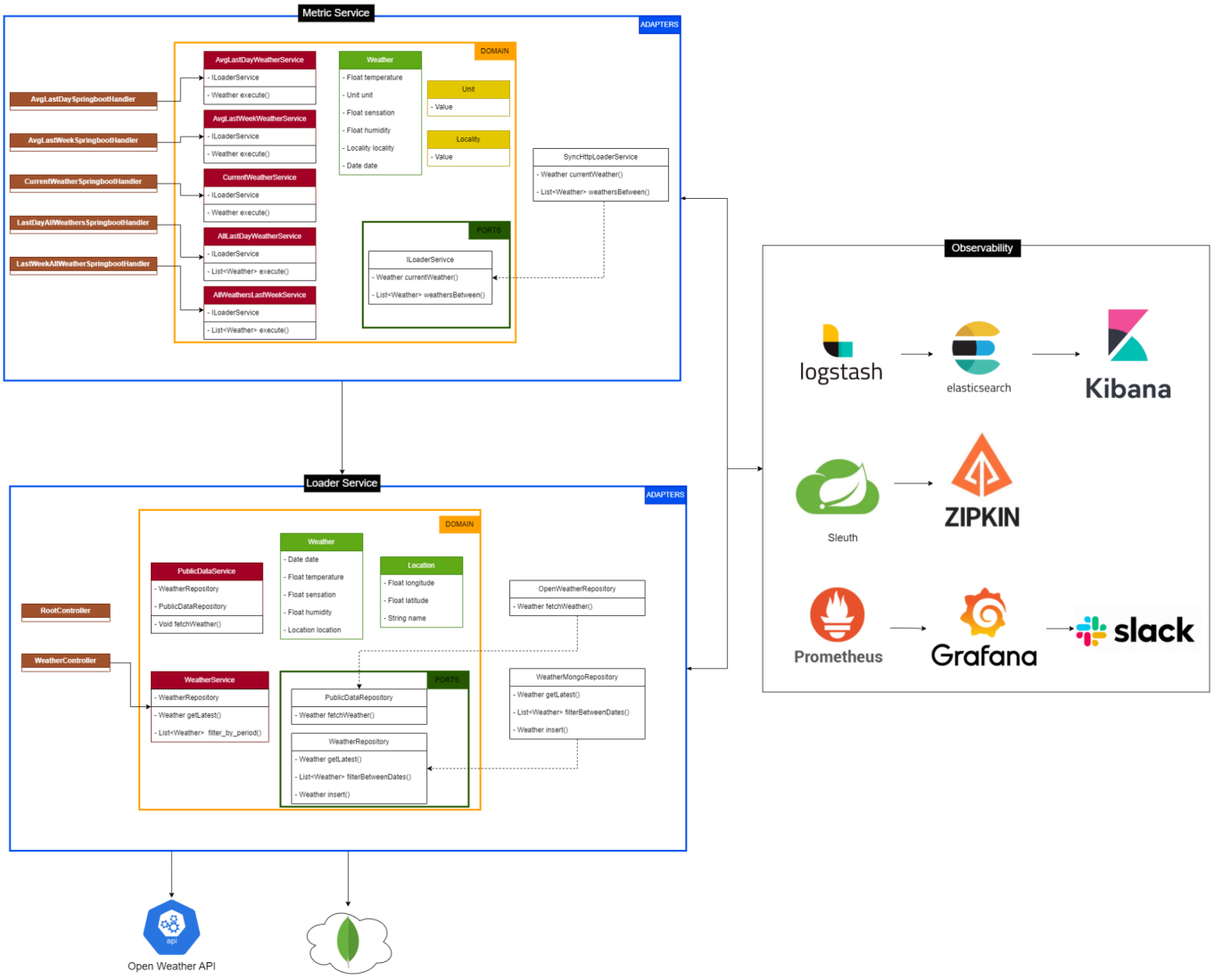
Caso de uso	CU-3
Título	Obtener climas de la ultima semana

Actor	Cliente
Descripción	Como cliente quiero obtener todos los climas registrados en la última semana.

Caso de uso	CU-4
Título	Obtener promedio del clima
Actor	Cliente
Descripción	Como cliente quiero consultar el clima promedio en las últimas 24 hs.

Caso de uso	CU-5
Título	Obtener el promedio del clima de la ultima semana
Actor	Cliente
Descripción	Como cliente quiero consultar el clima promedio de la última semana.

Diagrama de arquitectura



Stack tecnológico

I. Lenguajes

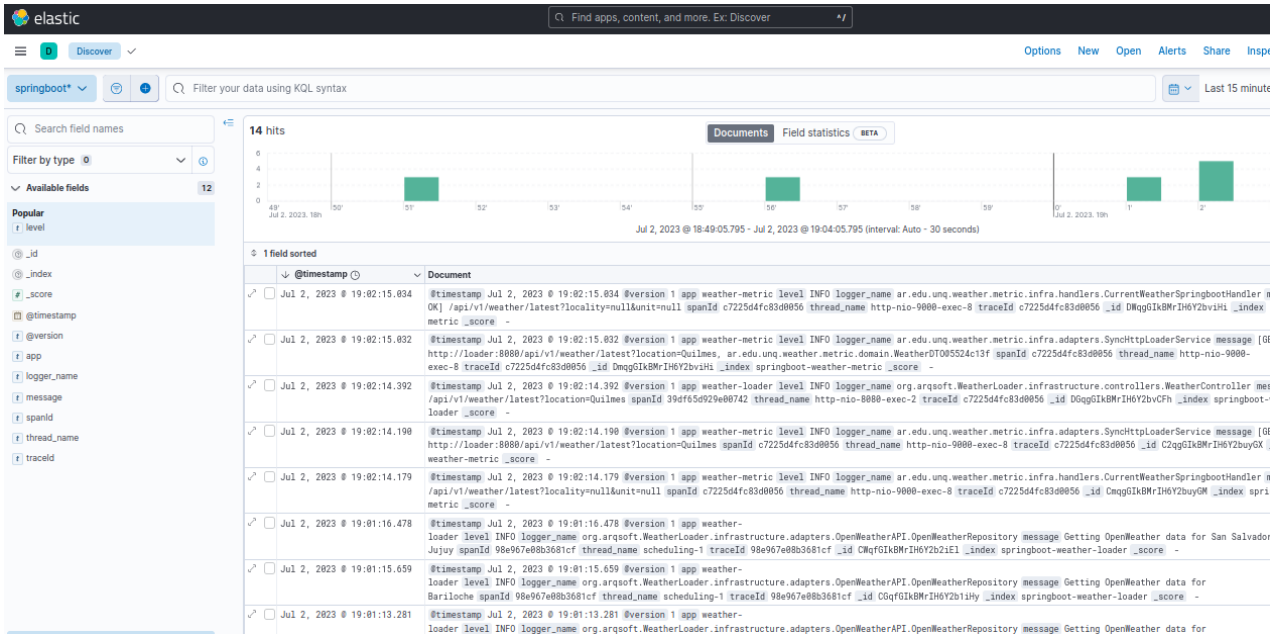
Java/Kotlin: se realizó el desarrollo del componente Loader en Java y el componente Metric en Kotlin. Debido a que Kotlin compila en la JVM de Java, podría interpretarse que su

utilización se basó en la velocidad de desarrollo que brinda por su acotación de código. Y en general, este lenguaje es uno de los más afianzados, por lo cual, para su desarrollo e investigación de herramientas de los requisitos de este trabajo nos redujo la carga de búsqueda de dudas en comunidades y libros. Cabe recalcar que ambos servicios se desarrollaron utilizando el framework Springboot.

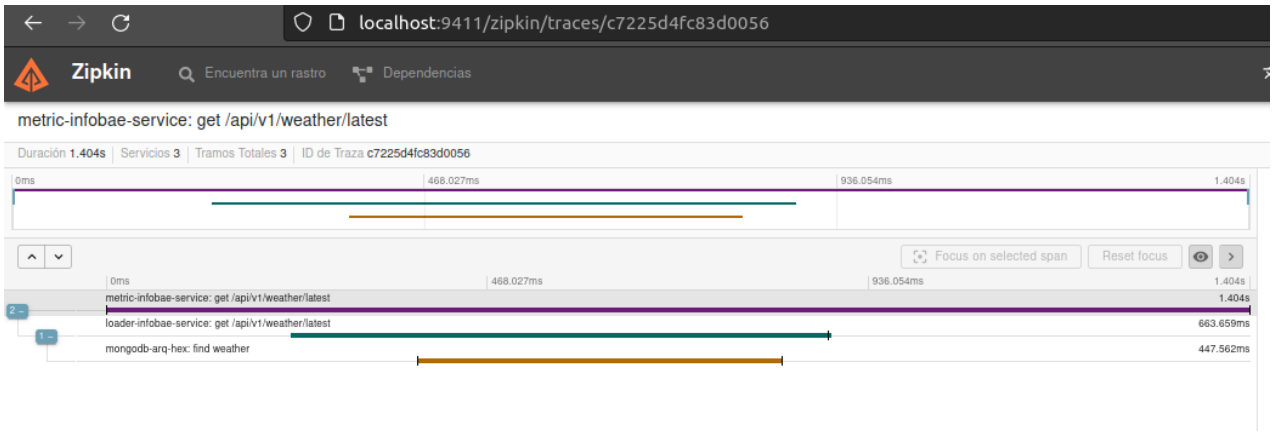
II. Observabilidad

Se implementaron todas las estrategias de observabilidad solicitadas con los siguientes stacks:

1. Log Aggregation: se utilizó el stack ELK, conformado por las herramientas ElasticSearch, como base de datos para logs; Logstash, como recompilador de logs de las aplicaciones; Kibana, como interfaz gráfica para visualizar los logs; y por último, SLF4J como generador de logs en los componentes desarrollados. Todo esto genera la centralizacion de logs y aumenta la capacidad y facilidad de analisis de los mismos.



2. Distributed Tracing: se desarrolló bajo la librería Zipkin en conjunto con Sleuth como herramienta de trazabilidad de una request, con la configuración correspondiente en cada servicio para enviar la información necesaria para realizar las trazas.



3. Metrics aggregation: se trabajó con la combinación de herramientas de Spring Actuator como recompilador de métricas en cada servicio, Prometheus como cliente de consulta de las métricas que genera el recompilador y Grafana como interfaz para visualizar en un dashboard las diferentes métricas de las aplicaciones.

4. Alerting: se configuraron alertas utilizando Grafana, gracias a las métricas que lee de Prometheus y en asociación a un WebHook de Slack, se logró tener un canal al cual Grafana envía alertas para que los desarrolladores estén informados de las situaciones marcadas para una asistencia manual.

III. Tolerancia a fallos

1. Timeout: se configuró con RestTemplate los time-outs de las consultas del componente Metric hacia el componente Loader, y del componente Loader hacia la api externa OpenWeather. Los timeouts configurados son de tres segundos.
2. Bulkhead: con Resilience4j se configuraron con el patron que utiliza por defecto, Semaforo, los threads maximos que el servicio de metricas puede utilizar concurrentemente. Se utilizaron varias configuraciones, la utilizada para los tests de carga finales fueron de hasta doscientos threads concurrentes y tiempo de espera para los recursos de 100ms en los endpoints que corresponden a consultas de el

clima actual, y cien llamadas concurrentes con 100ms de espera por el recurso para las llamadas que requieren un analisis semanal. Esta configuracion se realizo bajo determinar que los potenciales usuarios de esta aplicacion veran y estaran consumiendo en mucha mayor medida el clima actual o el de las ultimas horas que los que son mas de analisis, que a la vez, estos ultimos requieren un mayor procesamiento de informacion para obtener respuesta.

3. Circuit Break: se utilizó Resilience4j para configurar este patrón en las queries que parten del servicio de Metric hacia el de Loader. La configuracion para este patron fue variada, describimos una utilizada para los tests de carga en el endpoint del clima actual, que fue con un threshold del 50% de fallos para pasar de CLOSED a OPEN en un total de cada 1000 requests por un segundo se mantiene open y se transiciona automaticamente a HALF-OPEN donde se espera que de veinte llamadas quede por debajo del treshold y transicionar a CLOSED, o si lo superan de nuevo a OPEN.
4. Rate Limiter: con Resilience4j se configuro en ambos servidores para limitar la cantidad de llamados a un endpoint especifico.
5. Retry: se utilizó Resilience4j para configurar este patrón en las queries que parten del servicio de Metric hacia el de Loader. A su vez, la configuracion ignora los errores generados por requests como 404 o 400, ya que por mas requests que se hagan no se podria obtener una respuesta favorable.

IV. Test de carga

Se utilizó JMeter como herramienta para crear y ejecutar las distintas pruebas de estrés al sistema.

V. Consumir API externa

Se utilizó Spring Schedule para configurar un servicio que, cada cinco minutos, realiza una tarea que consume la API de OpenWeather y extrae la información del clima atual de las distintas ciudades que fueron configuradas.

VI. Documentacion de APIs

Se generó la configuración de los servicios con Swagger.

VII. Dockerizacion

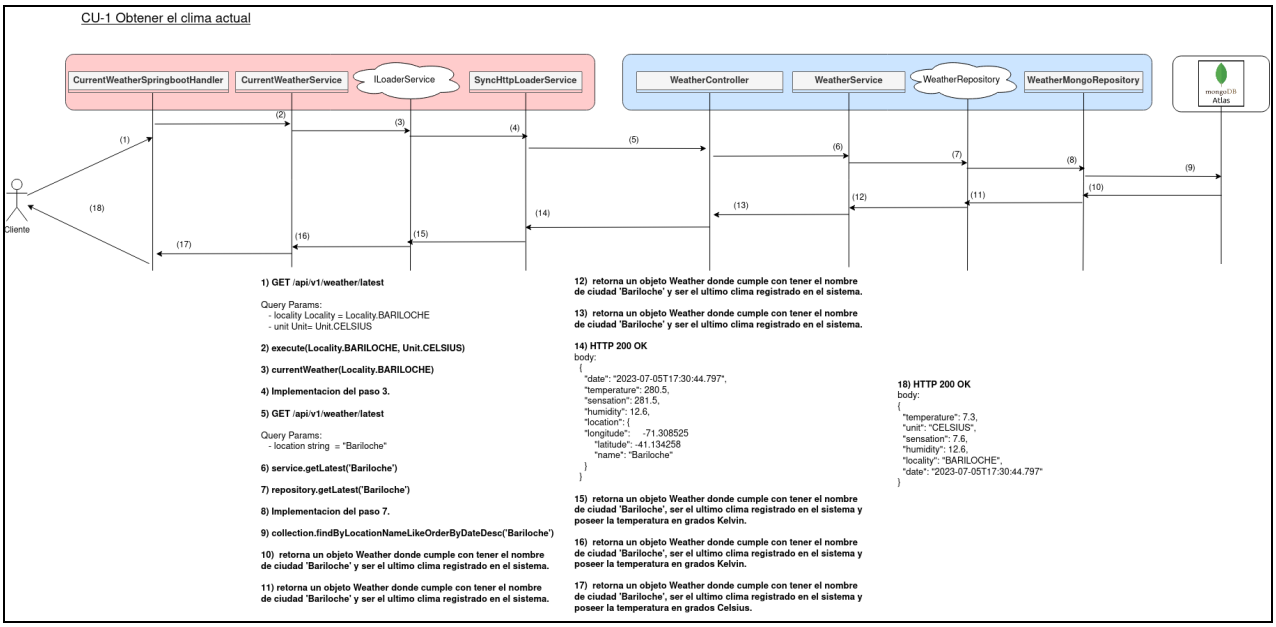
Se dockerizaron todos los servicios necesarios para no tener que instalar herramientas para quien desee probar la aplicación.

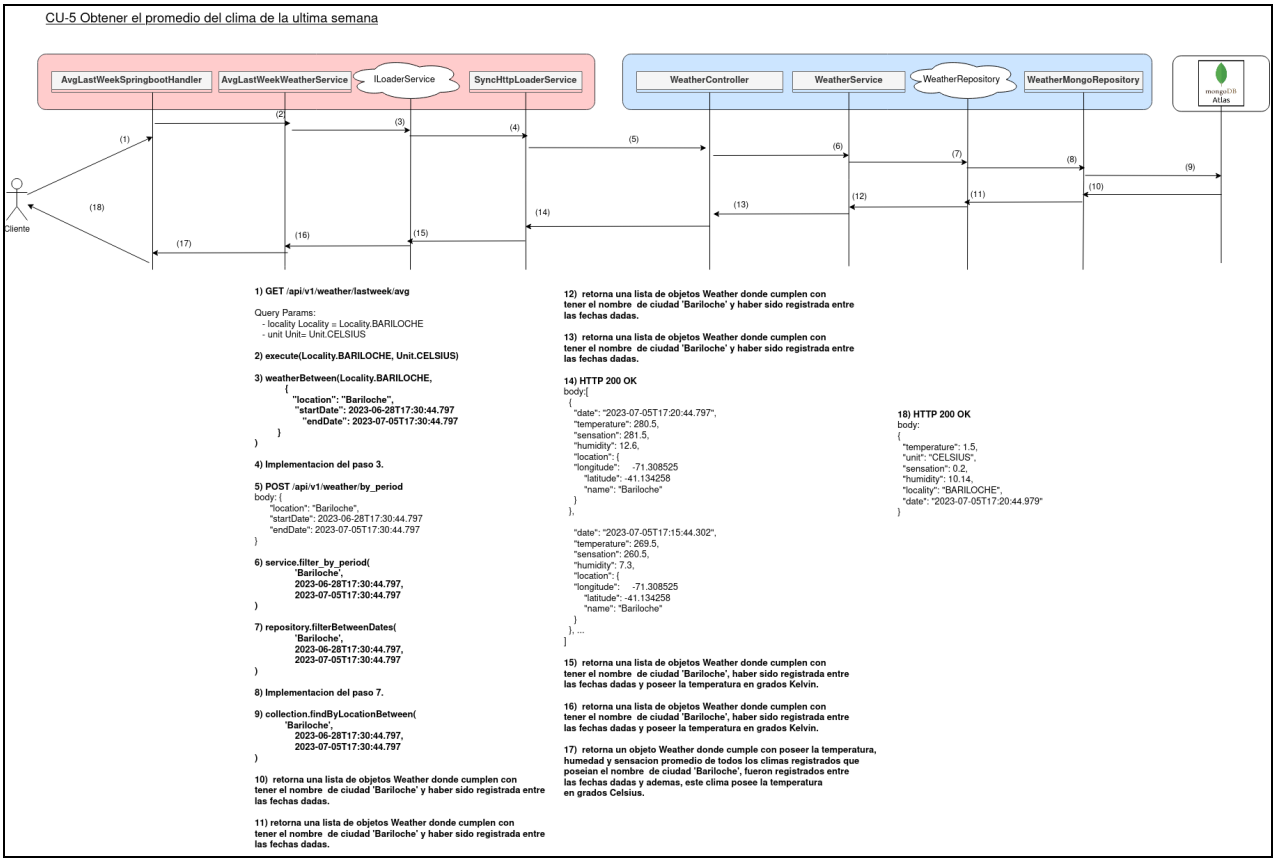
VIII. Persistencia

Se utilizó Mongo Atlas para persistencia de datos.

Diagramas de secuencia

Se documentaron los diagramas de secuencia del CU-1 y CU-5, debido a que luego las secuencias se vuelven redundantes porque son casi equivalentes a estas. Se pueden encontrar todos los diagramas de este trabajo en formato .png para una mejor visualizacion.





Alerting

Alto consumo de CPU

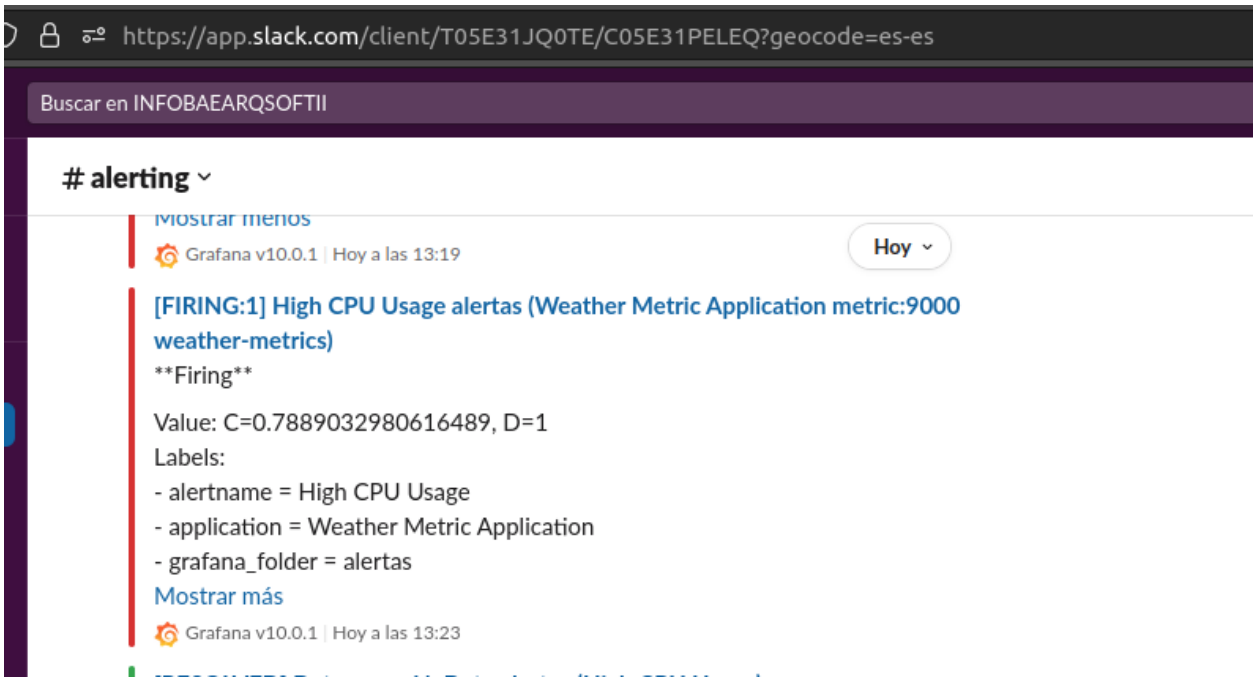
Esta alerta fue desarrollada y se pudo generar un entorno en el cual generarla.

Creemos que la configuracion de esta alerta es la base para cualquier sistema que se desee desarrollar porque es el cerebro de los servidores y uno necesita llevar un control riguroso sobre dicho componente. Se configuro que si al 70% de consumo de cpu en cualquier momento de los ultimos cinco minutos en cualquiera de los componentes, Grafana dispare la alerta.

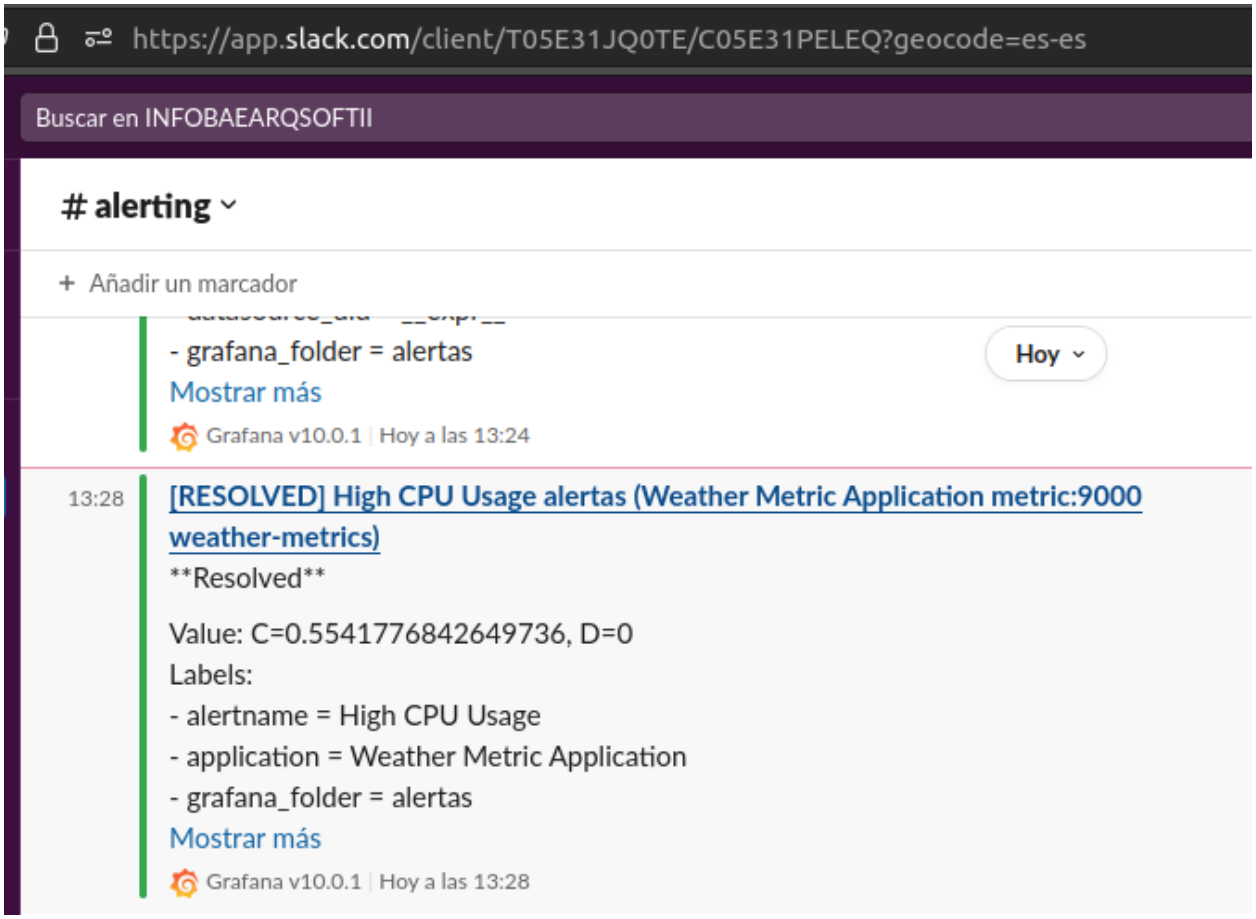
Utilizamos la interfaz grafica que provee grafana para la configuracion de la misma en la que, si los parametros de la alerta superan el threshold configurado, se encargara de enviar un mensaje a traves del WebHook de Slack a un canal que fue configurado previamente. La accion de monitorear las metricas del consumo de CPU se colocaron dos minutos, y este monitoreo dura un minuto, el cual mira los ultimos cinco minutos de la metrica en cuestion.

Desde Grafana podemos observar en las metricas del cpu, donde del lado del nombre vemos el estado del monitoreo, un corazon verde cuando los parametros estan normales, y un corazon rojo roto cuando se dispara la alerta. Al pasar el mouse sobre las flechas arriba del grafico con linea punteada, nos indica los resultados del monitoreo en ese momento, el cual, mientras no haya modificaciones en el estado del, en este caso el cpu, no se volvera a registrar uno, ya que indican los cambios.

Por otro lado, esta es la alerta que se dispara en slack a la hora de que la configuracion de la alerta indique que hay metricas que deben dispararse.



Por ultimo, cuando la situacion se normaliza, Grafana tambien envia una alerta de que el problema se soluciono y el servicio vuelve a estar dentro de los parametros normales.



Test de carga

Se utilizo JMeter como herramienta de testing para estresar el sistema como fue mencionado previamente.

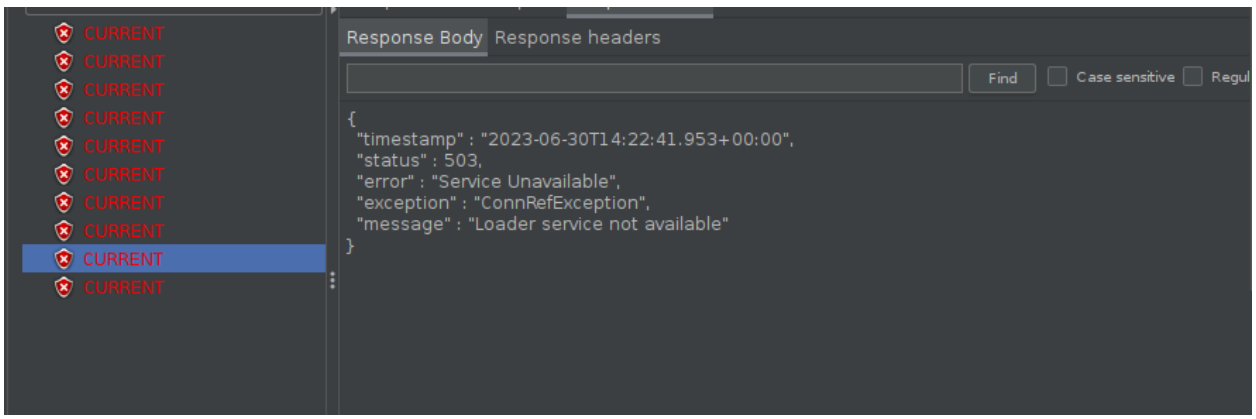
1. Primer prueba: ambiente controlado

Se realizaron peticiones a GET /api/v1/weather/latest, que por defecto retorna el clima actual en grados celsius de Quilmes. Este primer test de carga se realizo con el objetivo de obtener un ambiente controlado para eplicar las estrategias de resiliencia implementadas con bajos parametros, destacar que no se ha levantado al servicio de loader para simular que el servidor esta caido y a su vez se le coloco un delay de dos segundos en el endpoint para forzar la utilizacion de varios threads. La configuracion fue:

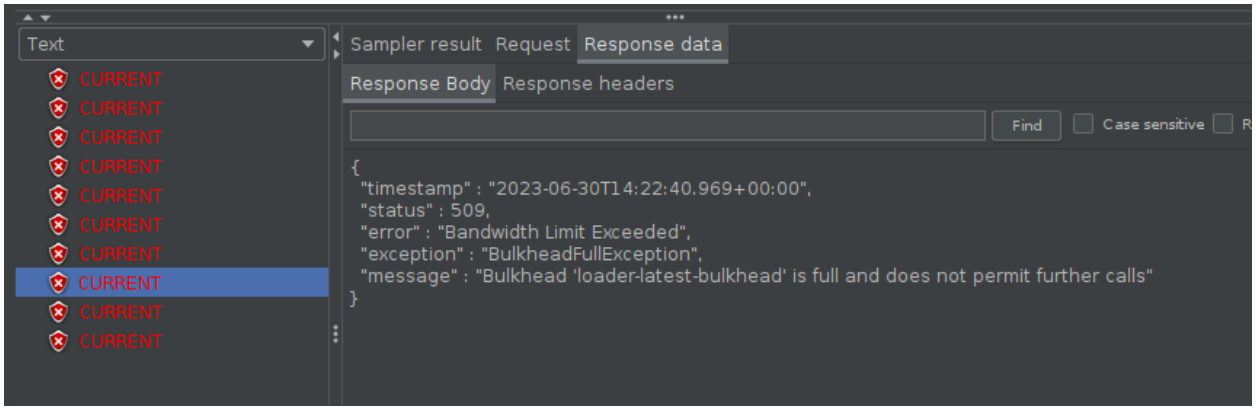
```yaml
resilience4j:
  circuitbreaker:
    instances:
      loader-service-circuit-breaker:
        slidingWindowSize: 10
        minimumNumberOfCalls: 5
        permittedNumberOfCallsInHalfOpenState: 3
        waitDurationInOpenState: 5s
        failureRateThreshold: 50
  retry:
    instances:
      loader-service-retry:
        maxRetryAttempts: 2
        waitDuration: 50ms
        retryExceptions:
          - org.springframework.web.client.RestClientException
          - ar.edu.unq.weather.metric.domain.exceptions.InfoBaeInternalServerError
  bulkhead:
    instances:
      loader-latest-bulkhead:
        maxConcurrentCalls: 2
        maxWaitDuration: 500ms
```

con lo que desde JMeter se configuraron 10 clientes que en un segundo consumieron dicho endpoint. Los resultados fueron:

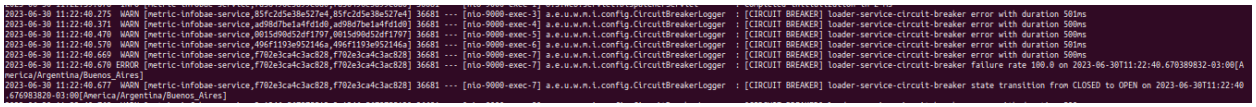
- Las primeras dos llamadas contestaron lo configurado en el sistema ante errores de conexion con el servicio de loader, 503 SERVICE_UNAVAILABLE



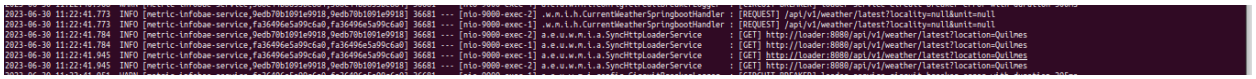
- Las siguientes ocho llamadas fueron respondidas con el fallback ante el error que levanta el patron Bulkhead cuando tiene la cantidad maxima de threads funcionando, con un status de 509: BANDWIDTH_LIMIT_EXCEEDED



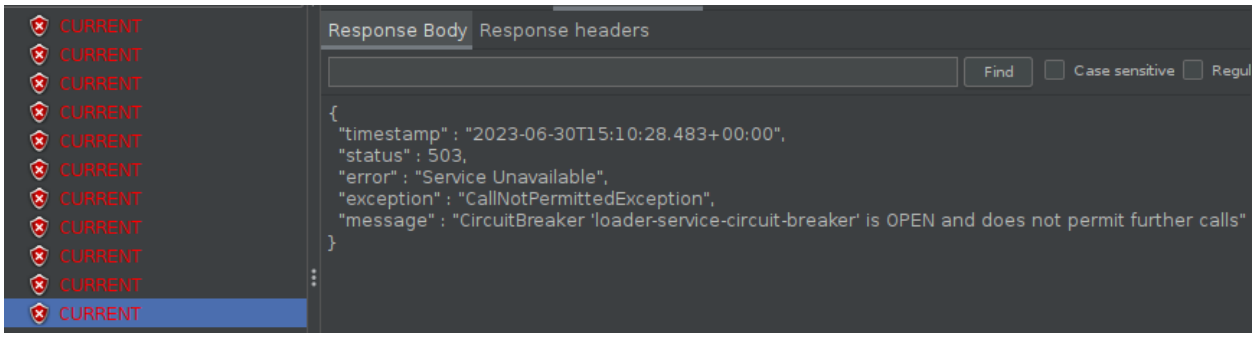
Se pudo observar como el circuit breaker fue iterando el error hasta superar el threshold y pasar a OPEN.



Tambien los retrys, que ante dos llamadas desde los handlers, el adapter intento obtener una respuesta positiva un total de dos veces por cada llamada (los threads fueron exec-1, exec-2).



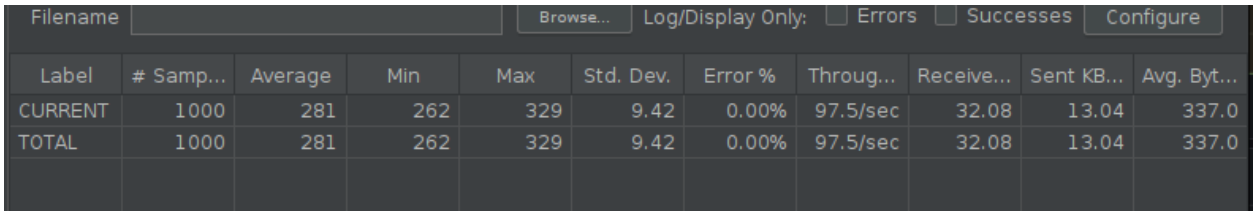
Por ultimo, realizando una siguiente llamada podemos observar que el status es un fallback con un nuevo mensaje, indicando que el estado del circuito es OPEN.



2. Segunda prueba: componentes listos

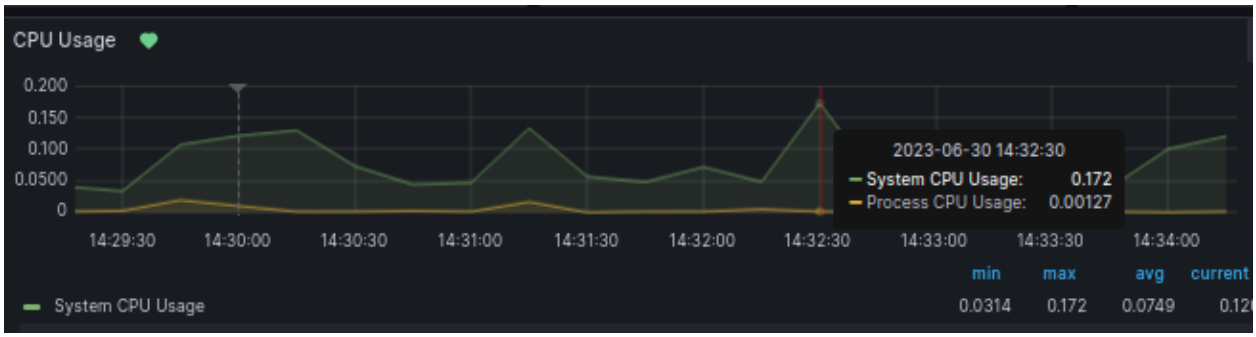
Para esta prueba ya se utilizaron ambos componentes de la aplicacion. El test fue sobre el mismo endpoint que el anterior, pero esta vez se utilizaron 1000 clientes en diez segundos. Donde el sistema no tuvo errores.

- El resultado de las mil consultas

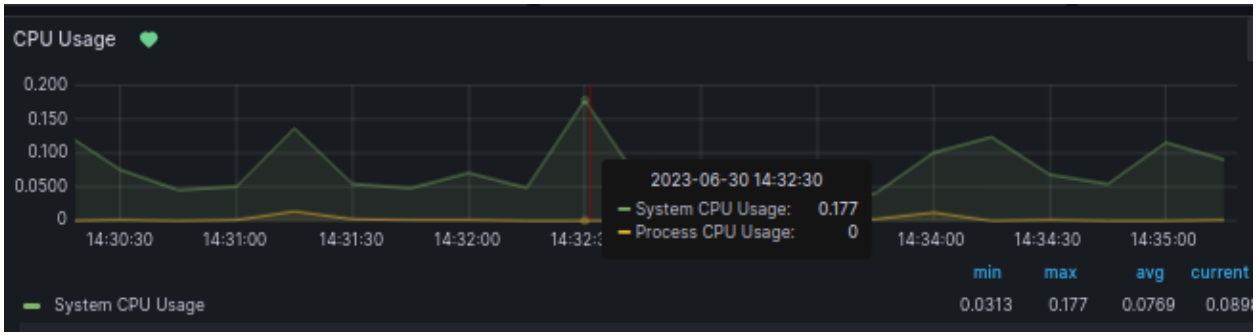


Label	# Samp...	Average	Min	Max	Std. Dev.	Error %	Throug...	Receive...	Sent KB...	Avg. Byt...
CURRENT	1000	281	262	329	9.42	0.00%	97.5/sec	32.08	13.04	337.0
TOTAL	1000	281	262	329	9.42	0.00%	97.5/sec	32.08	13.04	337.0

- El pico del cpu del loader



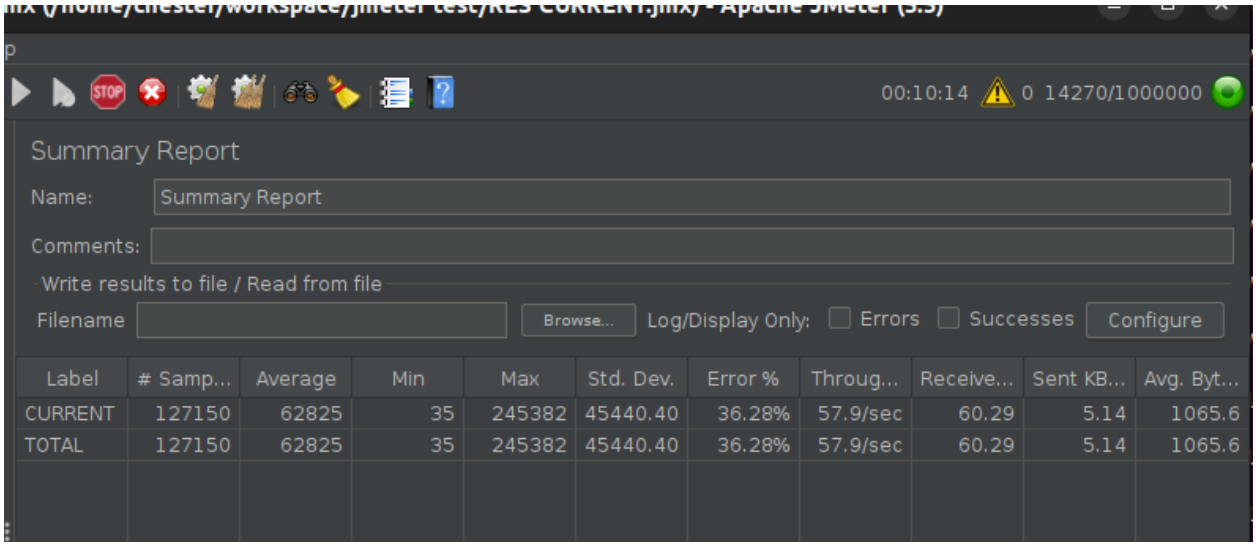
- El pico del cpu del metric



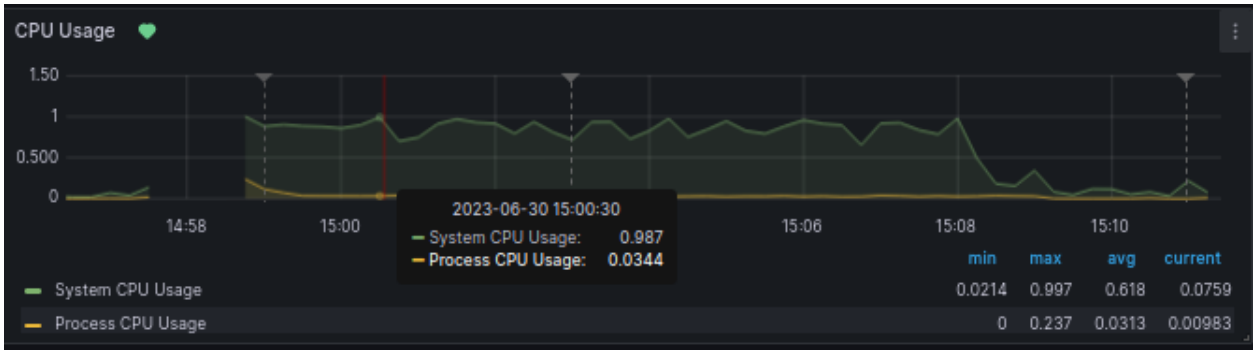
3. Tercer prueba: 200.000 usuarios sin resiliencia

Para esta prueba se quitaron las estrategias de resiliencia aplicadas en el proyecto para ver como se comporta el proyecto sin las mismas.

Luego de 100.000 peticiones en 10 min, dimos por finalizada la demostracion debido a que se volvia dificil mantener la computadora en este estado. Los errores fueron por timeouts que poseen automaticamente los threads de JMeter.



El CPU del loader fue consumido hasta en un 98% y disparo la alerta de grafana.



Por otro lado, el servidor de metric dejo de enviar metricas a Prometheus hasta no volver a recuperarse, pero al estar dockerizado y compilado en el mismo lenguaje, imaginamos una situacion similar.

4. Cuarta prueba: 10.000 usuarios con estrategias de resiliencia acotadas

Esta prueba se hizo para ver que tan rapido se podia atender clientes con una configuracion muy baja por parte de las estrategias de resiliencia.

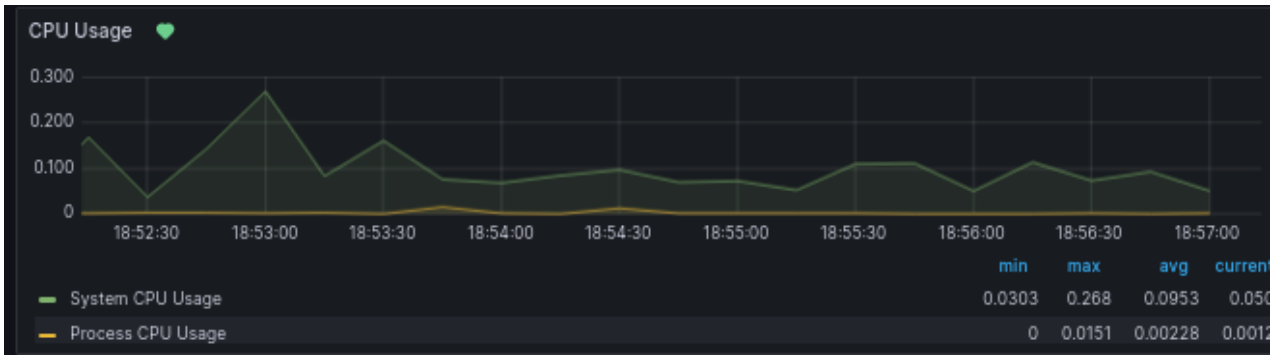
```
circuitbreaker:
 instances:
   loader-latest-circuit-breaker:
     registerHealthIndicator: true
     slidingWindowSize: 100
     minimumNumberOfCalls: 50
     permittedNumberOfCallsInHalfOpenState: 6
     automaticTransitionFromOpenToHalfOpenEnabled: true
     waitDurationInOpenState: 1s
     failureRateThreshold: 50
retry:
 instances:
   loader-latest-retry:
     registerHealthIndicator: true
     maxRetryAttempts: 1
     waitDuration: 20ms
     retryExceptions:
       - org.springframework.web.client.RestClientException
       - ar.edu.unq.weather.metric.domain.exceptions.InfoBaeInternalServerError
bulkhead:
 instances:
   loader-latest-bulkhead:
     maxConcurrentCalls: 100
     maxWaitDuration: 100ms
ratelimiter:
 metrics.enabled: true
 instances:
   loader-latest-rate:
```

```
    register-health-indicator: true
    limit-for-period: 100
    limit-refresh-period: 1s
    timeout-duration: 10ms
```

- En tan solo 22 segundos se respondieron todas las consultas con el siguiente resultado, que a su vez las 10.000 request fueron configuradas para enviar la mayor cantidad en el menor tiempo posible:

Label	# Sa...	Avera...	Median	90% L...	95% L...	99% L...	Min	Maxim...	Error %	Throu...	Recei...	Sent ...
CURR...	10000	200	14	555	1051	2003	2	4437	83.23%	369.4...	136.10	0.00
TOTAL	10000	200	14	555	1051	2003	2	4437	83.23%	369.4...	136.10	0.00

Si bien es un porcentaje alto de errores, todos fueron generados sin sobrecargar el sistema, ya que solo se exigio el cpu por debajo del 30%



5. 10.000 de usuarios con estrategias de resiliencia

Para esta prueba se utilizo la siguiente configuracion:

```
circuitbreaker:
 instances:
   loader-latest-circuit-breaker:
     registerHealthIndicator: true
     slidingWindowSize: 100
     minimumNumberOfCalls: 50
     permittedNumberOfCallsInHalfOpenState: 5
     automaticTransitionFromOpenToHalfOpenEnabled: true
     waitDurationInOpenState: 500ms
     failureRateThreshold: 50
```

```
retry:
 instances:
   loader-latest-retry:
     registerHealthIndicator: true
     maxRetryAttempts: 1
     waitDuration: 2ms
     retryExceptions:
       - org.springframework.web.client.RestClientException
       - ar.edu.unq.weather.metric.domain.exceptions.InfoBaeInternalServerError
bulkhead:
 instances:
   loader-latest-bulkhead:
     maxConcurrentCalls: 200
     maxWaitDuration: 100ms
ratelimiter:
 metrics.enabled: true
 instances:
   loader-latest-rate:
     register-health-indicator: true
     limit-for-period: 1000
     limit-refresh-period: 2s
     timeout-duration: 100ms
```

Los resultados mejoraron en gran medida, ya que solo hubo un 0,3% de errores generados durante un minuto tres segundos de duracion de la prueba.

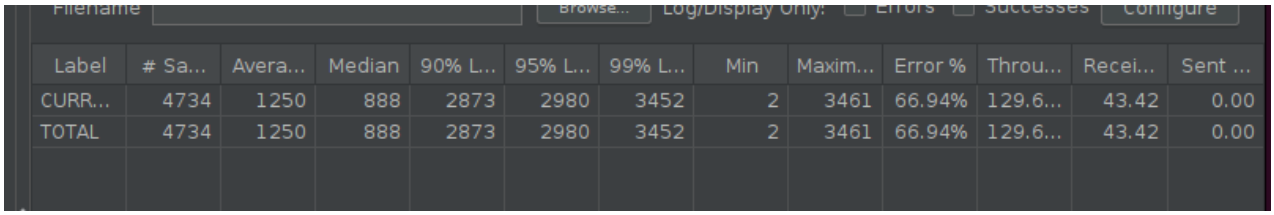
Label	# Samp...	Average	Min	Max	Std. Dev.	Error %	Throug...	Receive...	Sent KB...	Avg. Byt...
CURRENT	10000	296	2	1023	75.16	4.89%	49.9/sec	16.63	6.68	341.1
TOTAL	10000	296	2	1023	75.16	4.89%	49.9/sec	16.63	6.68	341.1

A diferencia del test sin estrategias de resiliencia, esta vez, solo se ha exigido el CPU al 46% maximo gracias a la resiliencia. Los errores fueron por Bulkheads, rate-limiter y Circuit Breaker.

6. Ultima prueba: 200.000 de usuarios con estrategias de resiliencia

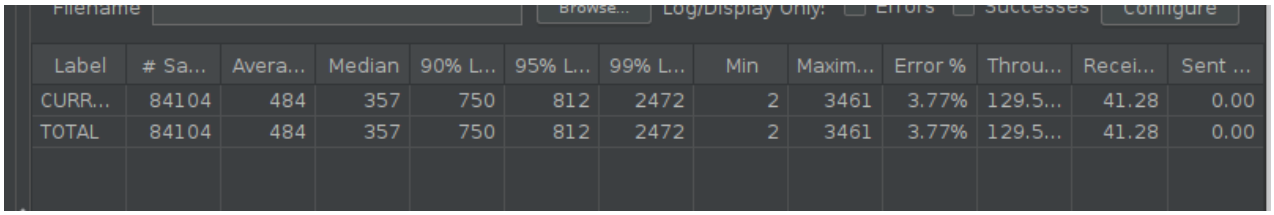
Luego de varias pruebas y errores mientras aprendimos a utilizar la herramienta de prueba y a su vez configurabamos todas las estrategias de tolerancia a fallos. Para esta prueba se configuraron los mismos parametros de prueba que en la que no tenia resiliencia.

- A los 30 segundos, se encontraron estas estadisticas:



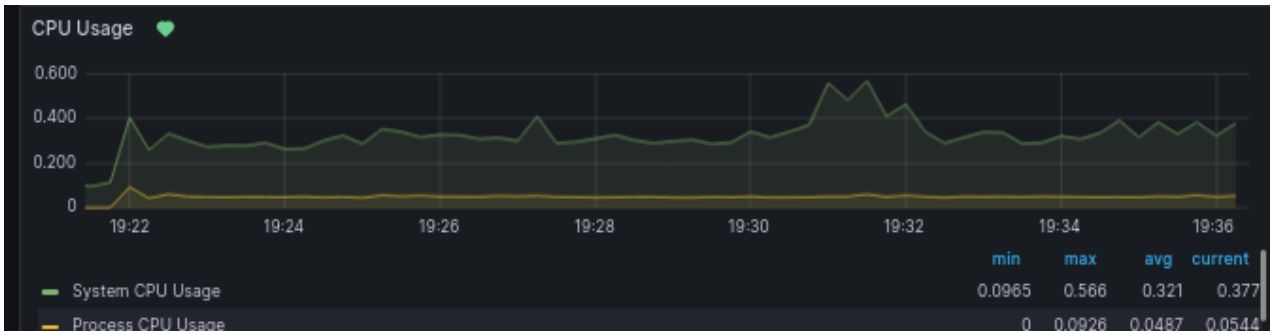
Label	# Sa...	Avera...	Median	90% L...	95% L...	99% L...	Min	Maxim...	Error %	Throu...	Recei...	Sent ...
CURR...	4734	1250	888	2873	2980	3452	2	3461	66.94%	129.6...	43.42	0.00
TOTAL	4734	1250	888	2873	2980	3452	2	3461	66.94%	129.6...	43.42	0.00

- Luego, a los diez minutos de la prueba se registraron estas estadisticas. Se respondieron menos request que en la prueba (3), pero con una taza muchisimo mayor. La baja de cantidad de respuestas estimamos que se debieron a los tiempos de espera que tiene configurado tanto el retry, los bulkheads y el rate limiter.

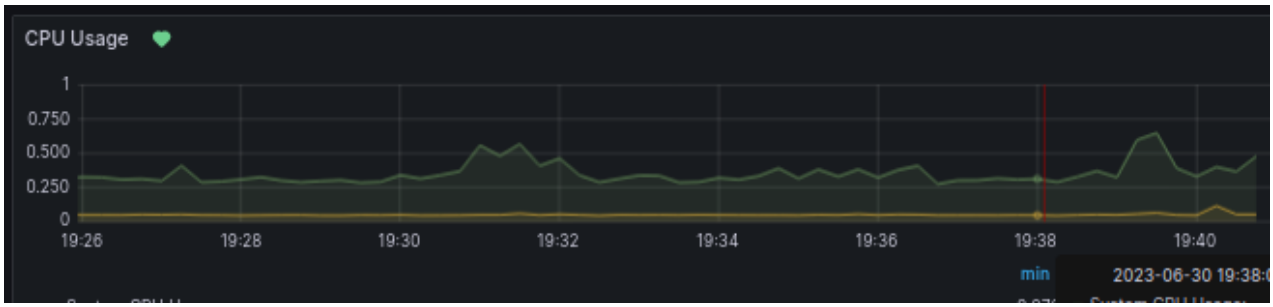


Label	# Sa...	Avera...	Median	90% L...	95% L...	99% L...	Min	Maxim...	Error %	Throu...	Recei...	Sent ...
CURR...	84104	484	357	750	812	2472	2	3461	3.77%	129.5...	41.28	0.00
TOTAL	84104	484	357	750	812	2472	2	3461	3.77%	129.5...	41.28	0.00

En este punto, se comprobaron las metricas desde Grafana, y la exigencia del CPU no alcanzo el 60%, con un promedio de alrededor del 40%.

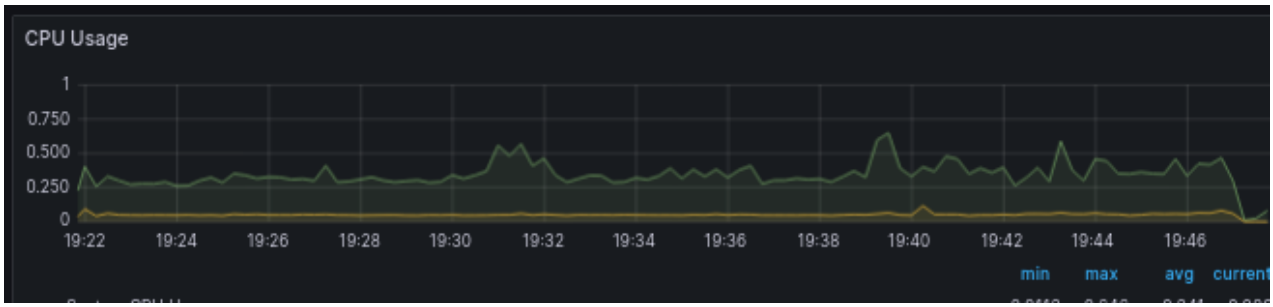


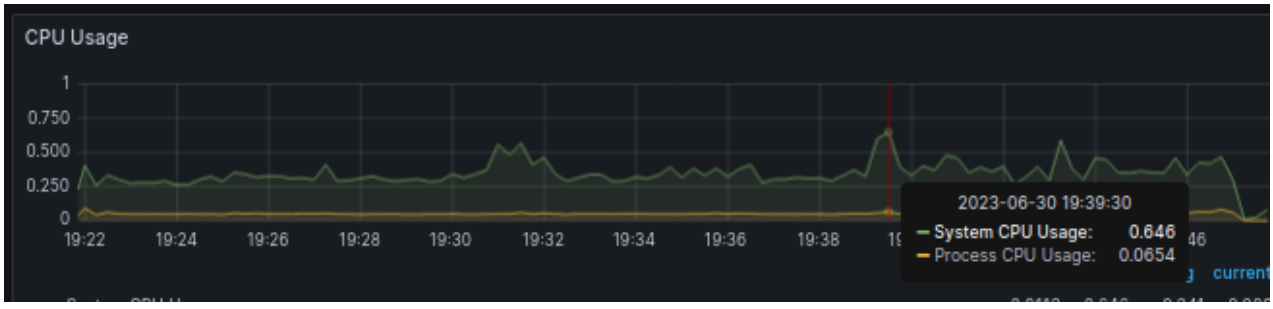
A los quince minutos, las metricas desde Grafana hicieron un nuevo pico de 66%, pero las respuestas siguieron siendo positivas y la computadora era totalmente utilizable.



Label	# Sa...	Avera...	Median	90% L...	95% L...	99% L...	Min	Maxim...	Error %	Throu...	Recei...	Sent ...
CURR...	140823	459	346	728	782	2006	2	3461	2.25%	129.1...	41.13	0.00
TOTAL	140823	459	346	728	782	2006	2	3461	2.25%	129.1...	41.13	0.00

Finalmente, al terminar el test en 23 minutos, Grafana refleja lo siguiente:





Un pico maximo de que no supera el 65% de utilizacion del CPU, y un promedio aproximado del 50%.

Por otra parte, el test:

Label	# Sa...	Avera...	Median	90% L...	95% L...	99% L...	Min	Maxim...	Error %	Throu...	Recei...	Sent ...
CURR...	200000	729	429	927	2053	8204	2	15026	2.03%	132.0...	42.06	0.00
TOTAL	200000	729	429	927	2053	8204	2	15026	2.03%	132.0...	42.06	0.00

Con estos numeros, podemos concluir que aplicar estrategias de resiliencia a un sistema se terminan traduciendo en una optimizacion a la hora de utilizar recursos del servidor, mejorando la performance y la salud, porque al nunca haberse superado el threshold del alerting se obtiene un sistema que se mantendra hasta en momentos de alta exigencia.

Conclusiones y posibles mejoras.

Creemos que este trabajo nos introdujo en el mundo de la observabilidad y tolerancia a fallos a nivel implementativo, porque ninguno de los dos integrantes del grupo habia trabajado con esto antes, lo cual nos genero un gran desafio a nivel equipo de estudiar e investigar cada herramienta utilizada, asi como tambien como eran sus configuraciones.

Tuvimos inconvenientes con las versiones del Framework y las herramientas utilizadas, por lo que al desafio tecnico que implicaba esta entrega se le sumo la realizacion de un downgrade y ajuste de versionado de librerias de herramientas.

Finalmente concluimos que estas estrategias implementadas ayudan a obtener un mejor contexto de la aplicacion, para el caso de la observabilidad, y a mantener una aplicacion saludable gracias a las estrategias de resiliencia que, a su vez, tener una configuracion eficiente al combinar las estrategias de tolerancia a fallos no es un trabajo sencillo.

Creemos que una buena mejora para este sistema serian dos:

- Por un lado, implementando y manteniendo una Request Cache en el servicio de metric para un menor consumo del endpoint del loader, y a la vez para mejorar la velocidad de respuestas a los clientes.
- Por el otro, la tarea de consumir la API externa en un componente separado del Loader porque si bien es una tarea asincrona, consume recursos de la maquina perteneciente al Loader. Por lo que creemos que un componente que consuma esa informacion y se la envie al loader ayudaria a desacoplar responsabilidades, respetando siempre el ownership de los datos.