

# Trabajo Práctico Final:

## Documentación

# Índice

[Índice](#)

[Tecnologías](#)

[Documentación de APIs](#)

[Diagramas de secuencia](#)

[Diagrama de arquitectura](#)

[Estrategias de Observabilidad](#)

[Log aggregation](#)

[Metrics aggregation](#)

[Distributed tracing](#)

[Alerting](#)

[Estrategia de Alerting](#)

[Tests de carga y dashboards de métricas](#)

[Time-out & Retries](#)

[Circuit Breaker](#)

[Fallback](#)

[Request caching](#)

[Adicional: Dashboard de temperaturas](#)

[Bibliografía](#)

# Tecnologías

Por simplicidad, dividimos las tecnologías seleccionadas en tres secciones: las relacionadas con la **infraestructura**, el **negocio**, las relacionadas con la **observabilidad y monitoreo**, y por último, las relacionadas con los **test de carga**. Como un factor común, todas las presentes eran conocidas por algún integrante del equipo.

Como **infraestructura**, se utilizó a [docker](#) y [docker-compose](#) como tecnología de infraestructura estándar para virtualización de contenedores, instalación y ejecución del sistema. Se la eligió al ser un estándar en la industria, y al ser portable por cualquier otro tipo de plataforma, ya sea **on-premise** o **cloud**.

Las tecnologías relacionadas a los servicios del **negocio** eran conocidas por el equipo para incrementar la agilidad del desarrollo. Estas fueron las tecnologías seleccionadas para este apartado con sus justificaciones particulares:

- [GoLang](#): Performante y simple. Además, se conocía una integración viable con **Prometheus** y **Grafana**.
- [MongoDB](#): Simple, alta disponibilidad y escalable. También, por su facilidad de tener queries performantes y flexibles, y la posibilidad de migrar a mongo atlas (plataforma cloud de mongo)
- [Redis](#): Escalable, alta disponibilidad y performante con su sistema de replicación. También por los límites que maneja al trabajar hasta con streaming de datos, por permitir tener estructuras de datos flexibles, y los servicios cloud disponibles en amplios proveedores.

Con respecto a la **observabilidad y monitoreo**, las tecnologías elegidas fueron por su fácil integración que ofrecen con **Prometheus** y **Grafana**. Se eligieron como pilares estas dos tecnologías mencionadas porque ambas son multiplataforma, open-source, fáciles de integrar con docker y docker-compose, poseen una basta documentación, herramientas y comunidad con varios lenguajes, y también, se complementan entre sí.

Volviendo al primer punto, esto es porque son desarrollados por ellos mismos o bien por terceros con el objetivo de integrarse principalmente a estas dos herramientas y ofrecer funcionalidades adicionales a las originales o que vienen por defecto.

- [Prometheus](#): Software para monitorear y alertar eventos. Persiste métricas en una time serie database (TSDB) alimentándose por llamadas HTTP. Posee queries flexibles y alertas en tiempo real. Este recopila dichas métricas utilizando la técnica “**scrape**”.

- **Grafana**: Aplicación web para visualizar analíticas, diagramas y gráficos. También, puede manejar alertas relacionados con los diagramas. Posee una fácil integración con **Prometheus**.
- **Loki**: Sistema de log aggregation provisto por **Grafana** y permite realizar queries a los logs.
- **Promtail**: Servicio que se encarga de extraer logs de los contenedores que posean un label arbitrario y establecido por el equipo (en nuestro caso, el label es “**logging: promtail**”). Es un agente que le provee los logs a la instancia privada de **Loki Grafana**.
- **Node-exporter**: Es una library de **Prometheus** que sirve para recolectar y exportar datos de infraestructura del host de docker.
- **Redis-exporter**: Es una library de terceros que exporta métricas del host de Redis para **Prometheus**.
- **Open Telemetry (OTel)**: Es un conjunto de herramientas, APIs y SDKs que sirven para instrumentar, recolectar y exportar datos de telemetría. Posee fácil integración con varios lenguajes (incluyendo **go**), **Prometheus** y **Grafana**.
- **Tempo**: Es una herramienta en el ecosistema de grafana por lo que permite ser integrado con este mismo, **Prometheus** y **Loki**. Es open-source y fácil de utilizar requiriendo únicamente un object storage para operar. Permite la ingesta de protocolos comunes de tracing como por ejemplo Jaeger, Zipkin y OpenTelemetry.
- **Blackbox**: Es una herramienta de **Prometheus** que en general permite sondear servicios utilizando HTTP, HTTPS, DNS; TCP y ICMP. Utilizado para obtener **Liveness probe** de los servicios.
- **Alertmanager**: Es una herramienta de **Prometheus**. Recibe alertas de **Prometheus**, las gestiona y las envía a distintos canales (Slack, email, Discord, Webhooks, Telegram, etc).
- **Cadvisor**: Recolecta métricas de infraestructura de los contenedores que están corriendo de Docker. Es una library provista por google.

Por último para realizar los **tests de carga**, se seleccionó a **Artillery** (<https://artillery.io/>). Esto es porque se instala fácilmente como una library de **nodeJS**, se definen los tests con archivos YAML, y se puede incluir lógica adicional al **processor** o al test en sí, utilizando **Javascript**.

## Documentación de APIs

Las APIs poseen cada uno su documentación con swagger en el host del servicio con el endpoint “**http://<host>:<port>/docs/index.html**”. En el Readme de cada servicio de negocio, se provee sus instrucciones para levantarlos independientemente de cada servicio. Además, se ofrece instructivo para instanciar el **docker-compose** en el repositorio de **weather-documentation**, haciendo

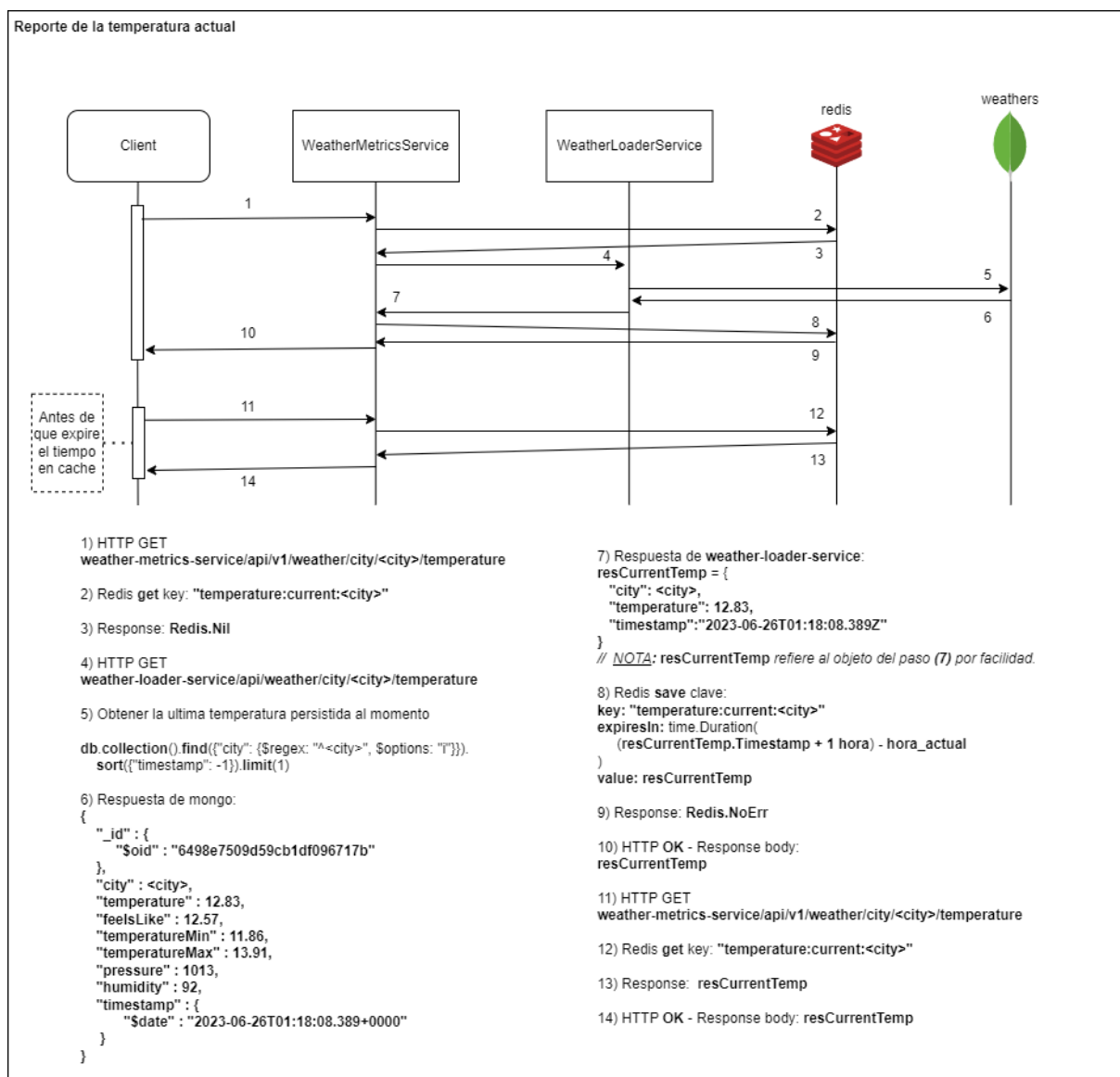
responsable al que lo quiere ejecutar de sustituir las variables de entorno con las credenciales de los servicios requeridos, sus hosts y puertos correspondientes a necesidad.

### Repositorios:

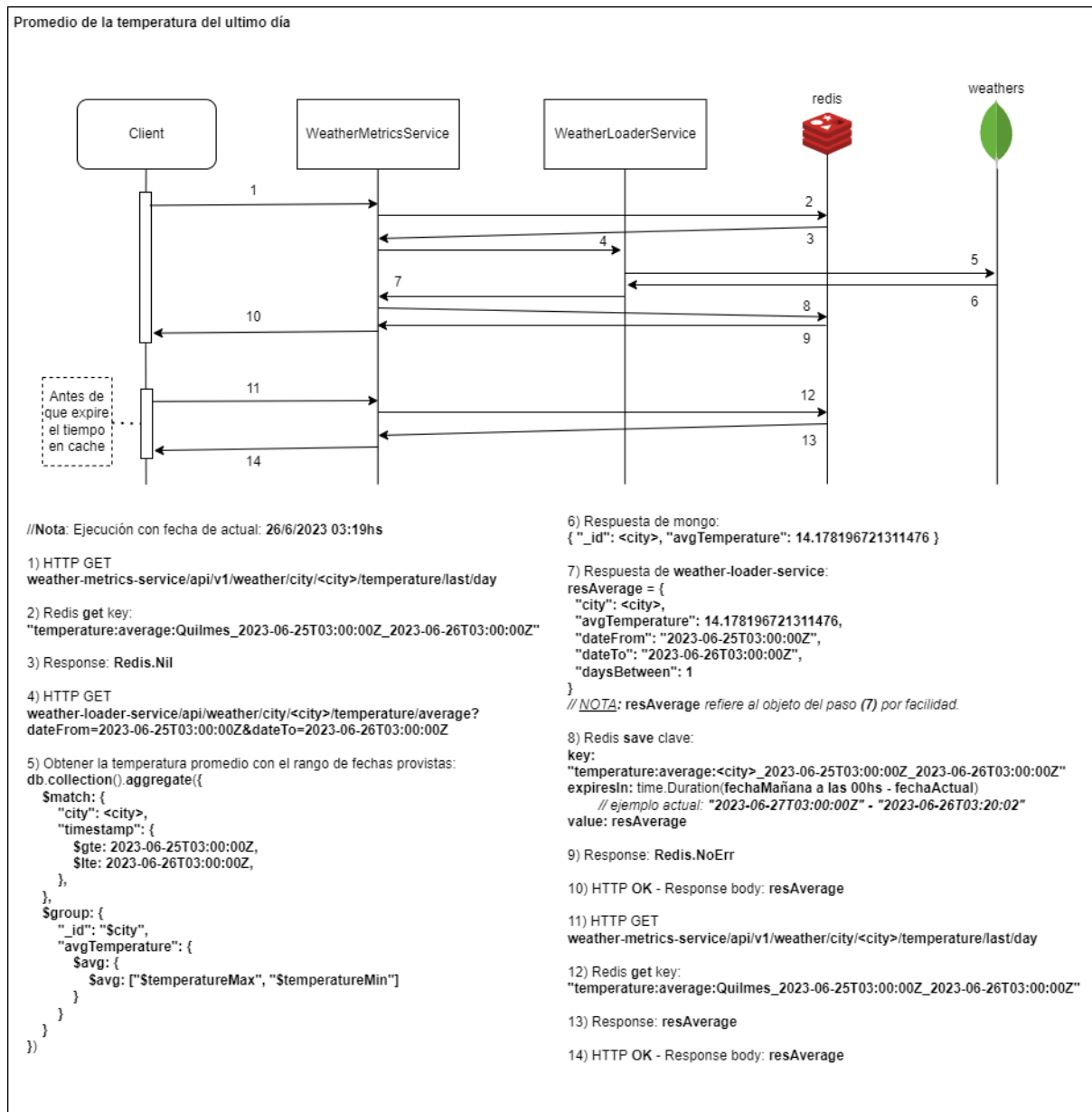
- **Weather-documentation:** [repositorio Github](#)
- **WeatherLoaderService:** [repositorio Github](#)
- **WeatherMetricsService:** [repositorio Github](#)

## Diagramas de secuencia

### Reporte de la temperatura actual

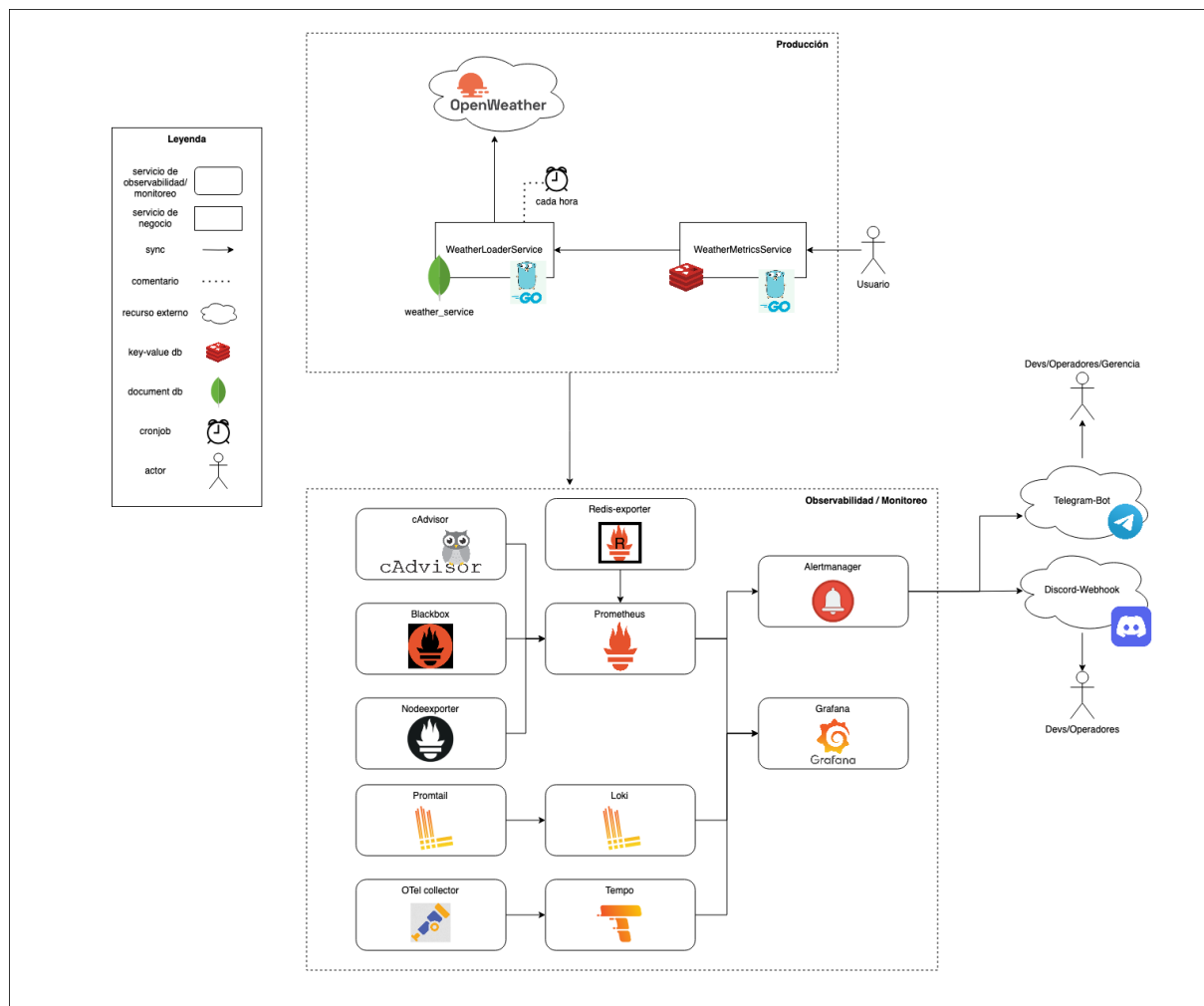


## Diagrama de temperatura promedio del último día



**Observación:** Este caso es idéntico al del caso de la temperatura promedio de la última semana. Esto es solo cambiando el rango de fechas de un día a una semana, en donde hay cualquier comunicación con fechas (**redis**, **weatherLoaderService** y **mongo**).

# Diagrama de arquitectura



## Estrategias de Observabilidad

Las estrategias se implementaron fundamentalmente en base con **Prometheus** y **Grafana**. Las tecnologías o herramientas adicionales giran en torno a estas dos mencionadas. Si se requiere más información o enlaces de alguna tecnología a la presente en esta sección, dirigirse a la sección de **tecnologías**.

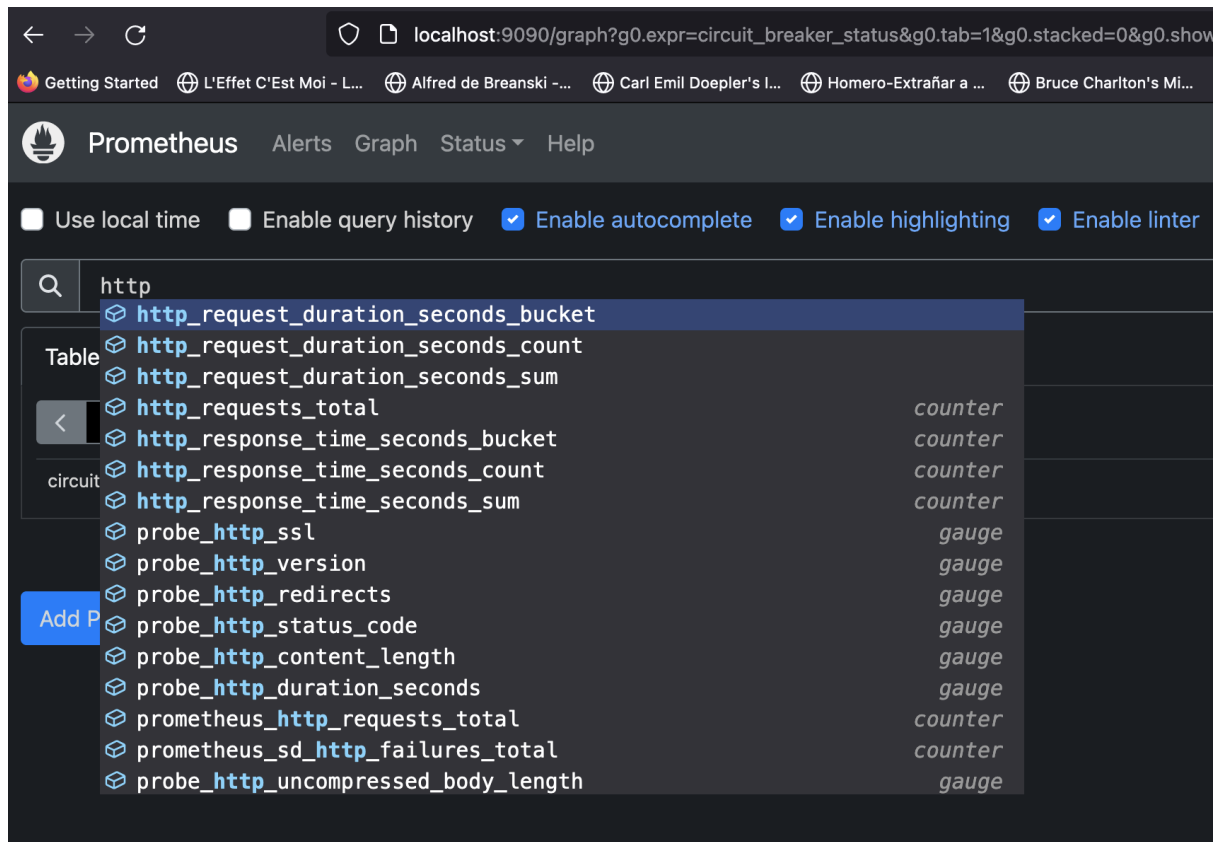
**Prometheus:** Sistema de monitoreo de código abierto. Este recopila métricas de distintos lugares con técnica de "**scrape**" y las almacena en una base de datos de series de tiempo.

En nuestro caso, configuramos los siguientes **scrapes**:

- node-exporter
- cadvisor
- prometheus
- blackbox

- weather-metrics-component (métricas de tráfico)
- tempo

## Captura de ejemplo UI Prometheus



## Ejemplo de scrape configs

```
# A scrape configuration containing exactly one endpoint to scrape.
scrape_configs:
- job_name: "nodeexporter"
  scrape_interval: 5s
  static_configs:
    - targets: ["nodeexporter:9100"]

- job_name: "cadvisor"
  scrape_interval: 5s
  static_configs:
    - targets: ["cadvisor:8080"]
```

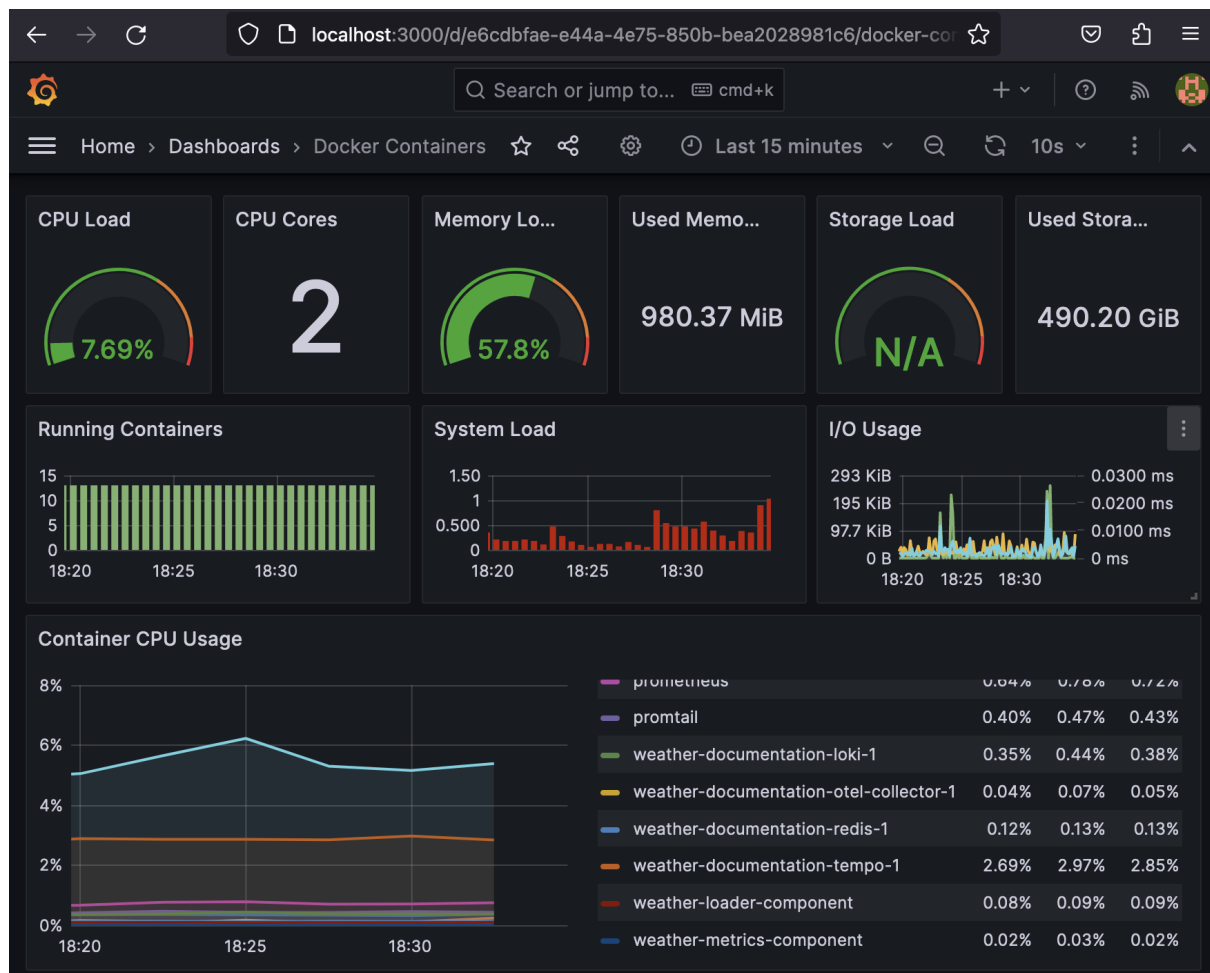
**Grafana:** Herramienta de visualización de datos de código abierto. Utiliza **Prometheus** como fuente de datos.

En nuestro caso, nuestras fuentes son:

- **Prometheus:** Métricas scrapeadas.
- **Loki:** Logs recolectados.
- **Tempo:** Trazas de requests.



## Captura de ejemplo Grafana UI



## Log aggregation

Implementado con **Loki**, **Promtail** y **Grafana**.

**Loki**: es como Prometheus pero para logs. Recolecta logs de distintos lugares y los almacena en una base de datos de series de tiempo.

**Promtail**: Agente de recolección de logs para enviar a Loki.

Servicio que se encarga de extraer logs de los contenedores que posean un label arbitrario y establecido por el equipo (en nuestro caso, el label es **"logging: promtail"**).

## Ejemplo configuración de Promtail

```
clients:
- url: http://loki:3100/loki/api/v1/push

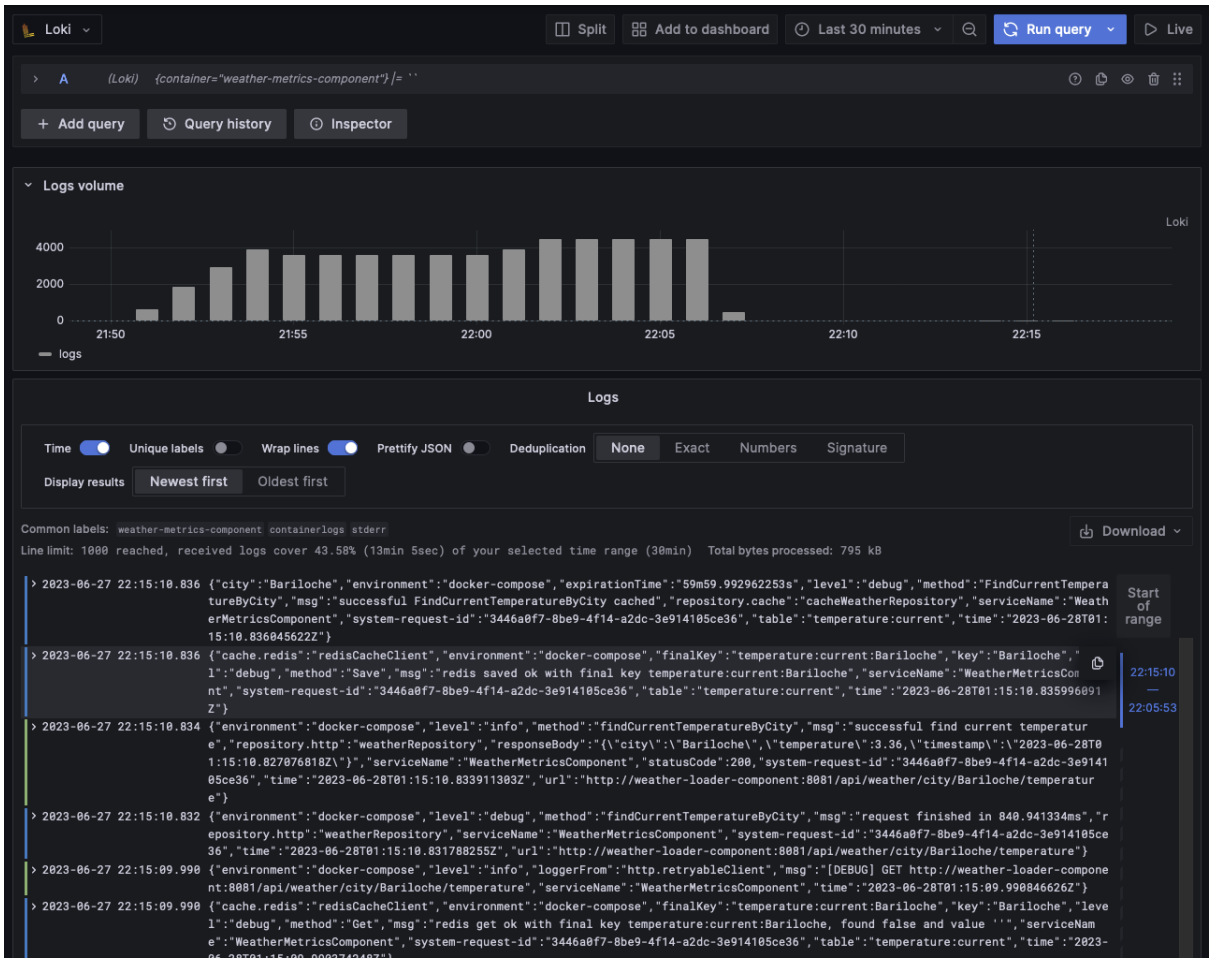
scrape_configs:
- job_name: flog_scrape
  docker_sd_configs:
  - host: unix:///var/run/docker.sock
    refresh_interval: 5s
    filters:
    - name: label
      values: ["logging=promtail"]
```

## Ejemplo de servicio con label agregado

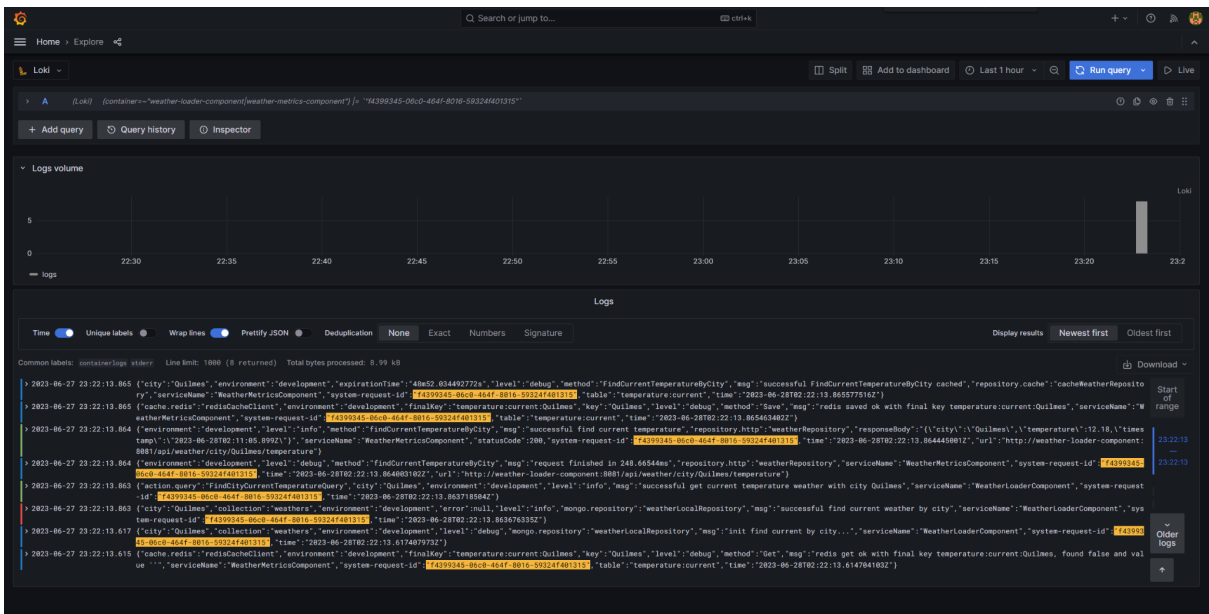
```
weather-metrics-component:

labels:
  org.label-schema.group: "weather-services"
  logging: "promtail"
  logging_jobname: "containerlogs"
```

# Ejemplo visualización de logs en Grafana con Loki como fuente



## Ejemplo de una búsqueda de logs de negocio por el “system-request-id” utilizando Loki y filtrando por los servicios deseados (weather loader y metrics components)



## Metrics aggregation

Implementado con **Prometheus**, **cAdvisor**, **node-exporter**, **redis-exporter** y **Grafana**.

**cAdvisor**: Recolecta métricas de los contenedores de Docker.

**node-exporter**: Recolecta métricas del Docker host.

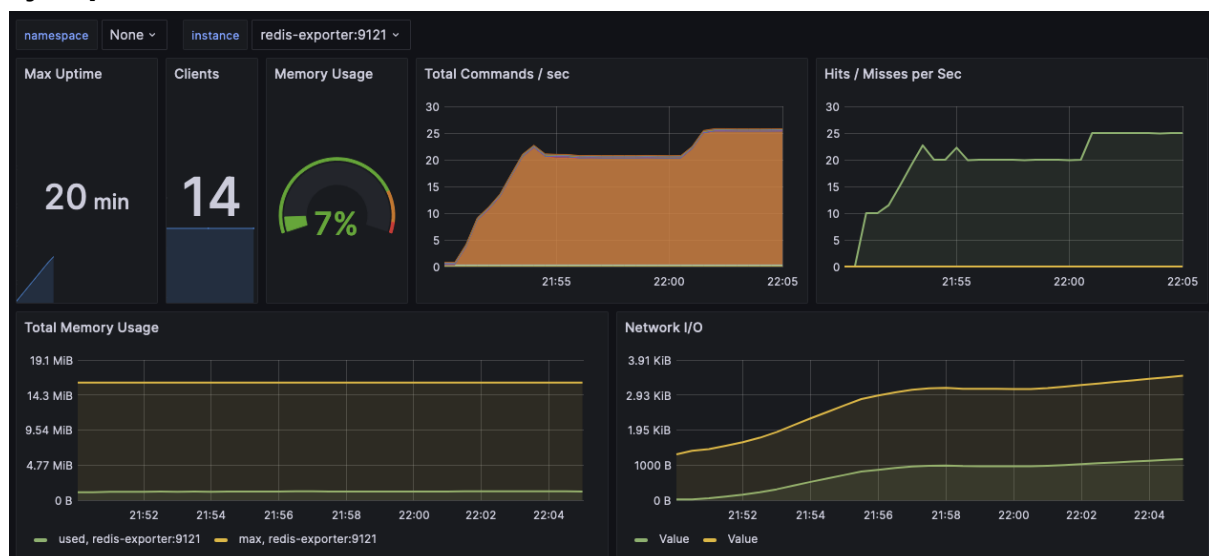
**redis-exporter**: Recolecta métricas de Redis.

Se agregaron **3** métricas custom al servicio de **Weather**:

- Requests totales por path y método.
- Suma total del tiempo tardado en responder requests (en segundos).
- Tiempos de respuesta por separado en "**buckets**".

Esas métricas son usadas manualmente para sacar alguna estadística y además se usan para dashboards armados en Grafana para obtener información mucho más importante.

## Ejemplo dashboard de Redis



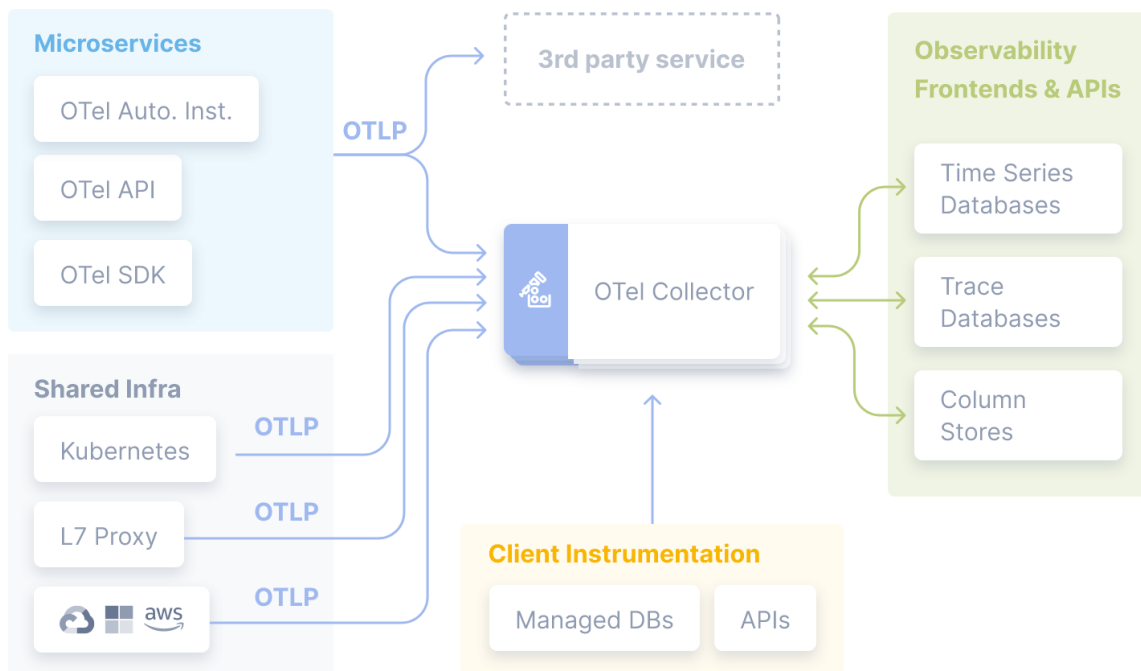
## Distributed tracing

Por un lado, se posee un `traceld` a nivel código para identificar las transacciones a nivel negocio con el campo “**system-request-id**”, esta se envía por **headers**.

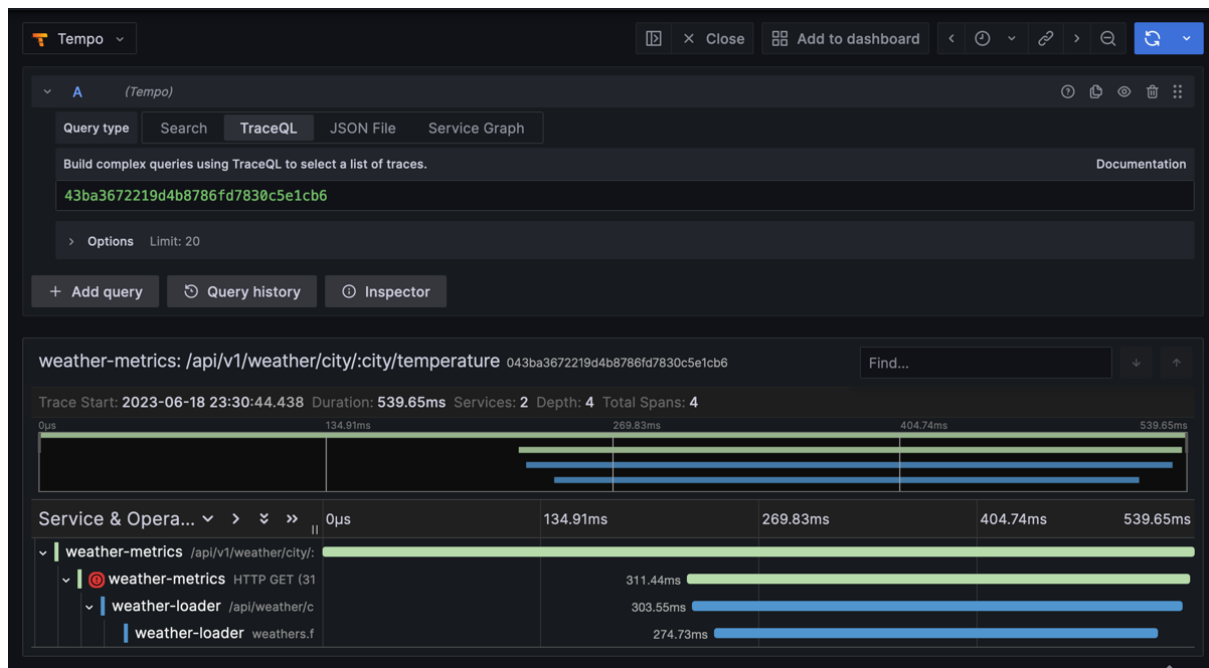
También, se integró **OTel** (Open Telemetry) para obtener una información más exhaustiva y detallada del ciclo de vida de un requests con sus interacciones a nivel de red (clientes de redis, http y mongo).

Implementado con libs de **OpenTelemetry** para Go, OTel Collector y Grafana-Tempo.

**OpenTelemetry**: Su objetivo es proveer un set de herramientas, APIs y librerías para enviar métricas a un backend de observabilidad.



**Grafana-Tempo**: Es un backend de trazas distribuido de código abierto. Soporta protocolos de trazas como Jaeger, Zipkin, y **OpenTelemetry**.



## Alerting

Implementado con **Prometheus**, **Blackbox Exporter** y **Alertmanager**. Luego, se puede agregar con **Grafana** más orientadas a **dashboards**, pero estas deben ser agregadas manualmente por la imposibilidad de importarlas y que sean configurables. Se tienen algunas creadas a modo de ejemplo, y una guía para su instanciación y configuración en el Readme del repositorio de documentación en la subcarpeta de `./documentación` y dentro de esta `./grafana_alerta_config`.

En la siguiente sección —**Estrategias de Alerting**—, se describe con más lujo de detalle todos los manejos de alertas, implementaciones, futuras implementaciones y simulaciones logradas.

**Blackbox Exporter:** Permite "sondear" distintos servicios utilizando HTTP, HTTPS, DNS, TCP y ICMP.

**Alertmanager:** Recibe alertas de **Prometheus**, las gestiona, y las envía a distintos canales (Slack, email, Webhooks, etc).

Configuramos un "scrape" de **Blackbox** en **Prometheus** para que sondee los servicios del Tiempo (hacer un GET de health-check para realizar el liveness-prove). En caso de no responder, se dispara una alerta en **Prometheus**. La alerta llega a

**Prometheus** y es administrada por **alert-manager**, que tiene configurado un **webhook** para enviar las alertas a **Discord**. Cada vez que se dispara una alerta (o hay una recuperación), se envía un mensaje al canal de Discord.

### Ejemplo configuración de Blackbox del módulo que devuelve “success” para los status 2xx

```
modules:
  http_2xx:
    prober: http
    timeout: 5s
    http:
      method: GET
```

### Ejemplo configuración Scrape Job en Prometheus para recolectar los datos de Blackbox

```
# checks "ping" status of the targets
- job_name: "blackbox"
  metrics_path: /probe
  scrape_interval: 5s
  params:
    module: [ http_2xx ]
  static_configs:
    - targets:
      - weather-loader-component:8081/
      - weather-metrics-component:8082/
```

### Ejemplo configuración de un receiver de Alert-Manager para notificar a Discord

```
receivers:
  - name: discord
    discord_configs:
      - webhook_url:
https://discord.com/api/webhooks/1117499416056705175/ihfuGM9SqPFGv
b5LudiGU_eIeS6Xsozemd5m99UIjSxa8Ij1Zj1bNvBnvr muc5i2vywK
```

# Estrategia de Alerting

## Estrategia general

Como estrategia general, se determinó la notificación de las alertas con severidad **warning** únicamente por los canales de **Discord**, y las de severidad **crítico** por todos los canales posibles (**Discord** y **Telegram**). Cada medio tiene la posibilidad de configurar sus **channels** en **Discord** o **grupos** en **Telegram** por donde se quiera notificar cada alerta. Esta estrategia se replican por las siguientes secciones.

Luego, a medida que se requiera se podría llegar a configurar una aplicación de pago (como [ops genie](#)) con esquemas de guardias. Con esto se puedan llevar a cabo rotaciones de personal donde se reciban todas las alertas categorizadas y divididas por equipo (por ejemplo, infraestructura con **operarios**, o bien, negocio con **desarrolladores** y/o **gerencia**), y que estas vayan escalando a sus superiores a medida que no son respondidas a tiempo con una ventana de **6hs-24hs** dependiendo del nivel de severidad y sus valores relacionados.

## Estrategia a nivel negocio

A nivel negocio, se utiliza principalmente el **apdex (Application Performance Index, o bien, índice de promedio de satisfacción respondiendo)** para alertar, ya que este índice es un estándar y se relaciona tanto con la satisfacción del usuario como con su experiencia con la aplicación.

Este índice va del rango de **0 a 1**, y tendrá como referencia que si su puntaje es de **1** tendremos a un **usuario satisfecho**, si el puntaje es de **0.5** a un **usuario tolerante** y un puntaje **0** a un **usuario totalmente frustrado**, o bien **insatisfecho**. Para nuestro negocio, definimos el puntaje promedio durante la última hora mínimo de **0.85** como un valor suficientemente estable, y el de **0.69** como un puntaje crítico.

**Apdex** se define con la siguiente fórmula:

$$\text{Apdex}_t = \frac{\text{SatisfiedCount} + (0.5 \cdot \text{ToleratingCount}) + (0 \cdot \text{FrustratedCount})}{\text{TotalSamples}}$$

en donde:

- **SatisfiedCount** son la cantidad de **requests** que tardan **menos de 0,5** segundo.



- **ToleratingCount** son la cantidad de **requests** que tardan **entre 0,5 y 1** segundo.
- **FrustratedCount** son la cantidad de **requests** que tardan **más de 1** segundo.
- **TotalSamples** es la cantidad **total** de **requests**.

Por lo tanto, se define que si el **apdex** tiene una media de puntaje **menor o igual al 0.85** durante **la última hora**, se empezará a enviar alertas con severidad **warning** al equipo para que se realice un análisis frente a la disminución de este índice. Si esta disminución de la media del **apdex** supera el mínimo de **0.69**, se enviará una alerta con severidad **crítica** donde se recibirá por todos los medios configurados para recibir dicha información.


Estas alertas se seguirán informando a medida que continúen en estos valores bajos sin superar dichos **mínimos**, ya que representan un riesgo para el negocio.

## Capturas de alertas simuladas de apdex por los canales actuales

### Alerta simulada de apdex crítico en canales Discord y Telegram

Arq Soft II - Alertas

4 members



Summary: Notification test

Details:

- alertname: *TestAlert*
- instance: *Grafana*

18:13

**ArqSoftII-Alerts**

✅ [RESOLVED] Apdex Score Below 0.69

Alert: Apdex Score Below 0.69

Values: [no value]

Severity: CRITICAL

Summary: In last hour, apdex average is below than minimum points (0.69)

Details:

- alertname: *Apdex Score Below 0.69*
- category: *business*
- grafana\_folder: *business*
- job: *weather-metrics-component*
- severity: *critical*

18:19

🔥 [FIRING] Apdex Score Below 0.69

Alert: Apdex Score Below 0.69

Values: Apdex promedio ultima hora=0.390625, Es inferior a 0.69 =1


Severity: CRITICAL

Summary: In last hour, apdex average is below than minimum points (0.69)

Details:


- alertname: *Apdex Score Below 0.69*
- category: *business*
- grafana\_folder: *business*
- job: *weather-metrics-component*
- severity: *critical*

18:21



📎 | Write a message...

# alertas\_business

Grafana  hoy a las 13:19

Resolved

Value: [no value]  
Labels:


- alertname = Apdex Score Below 0.69
  - category = business
  - grafana\_folder = business
  - job = weather-metrics-component
  - severity = critical

Annotations:

- grafana\_state\_reason = Updated
  - summary = In last hour, apdex average is below than minimum points (0.69)

Source: <http://localhost:3000/alerting/grafana/c02b5206-3492-499d-a64a-ccb3ab3f2975/view?orgId=1>  
Silence: [http://localhost:3000/alerting/silence/new?alertmanager=grafana&matcher=alertname%3DApdex+Score+Below+0.69&matcher=category%3Dbusiness&matcher=grafana\\_folder%3Dbusiness&matcher=job%3Dweather-metrics-component&matcher=severity%3Dcritical](http://localhost:3000/alerting/silence/new?alertmanager=grafana&matcher=alertname%3DApdex+Score+Below+0.69&matcher=category%3Dbusiness&matcher=grafana_folder%3Dbusiness&matcher=job%3Dweather-metrics-component&matcher=severity%3Dcritical)  
Dashboard: <http://localhost:3000/d/ZgBkHNhMz?orgId=1>  
Panel: <http://localhost:3000/d/ZgBkHNhMz?orgId=1&viewPanel=6>

**[RESOLVED] Apdex Score Below 0.69 business (business weather-metrics-component critical)**

 Grafana v9.5.2

Firing

Value: Apdex promedio ultima hora=0.390625, Es inferior a 0.69 =1  
Labels:

- alertname = Apdex Score Below 0.69
  - category = business
  - grafana\_folder = business
  - job = weather-metrics-component
  - severity = critical

Annotations:

Firing

Value: Apdex promedio ultima hora=0.390625, Es inferior a 0.69 =1  
Labels:


- alertname = Apdex Score Below 0.69
  - category = business
  - grafana\_folder = business
  - job = weather-metrics-component
  - severity = critical

Annotations:

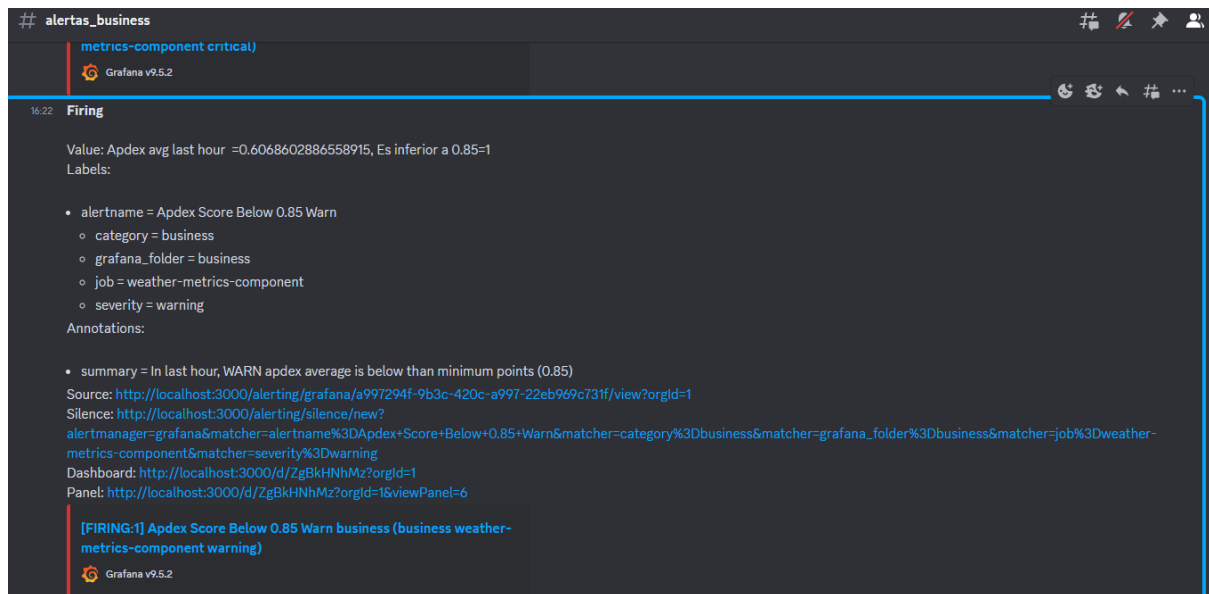
- summary = In last hour, apdex average is below than minimum points (0.69)

Source: <http://localhost:3000/alerting/grafana/c02b5206-3492-499d-a64a-ccb3ab3f2975/view?orgId=1>  
Silence: [http://localhost:3000/alerting/silence/new?alertmanager=grafana&matcher=alertname%3DApdex+Score+Below+0.69&matcher=category%3Dbusiness&matcher=grafana\\_folder%3Dbusiness&matcher=job%3Dweather-metrics-component&matcher=severity%3Dcritical](http://localhost:3000/alerting/silence/new?alertmanager=grafana&matcher=alertname%3DApdex+Score+Below+0.69&matcher=category%3Dbusiness&matcher=grafana_folder%3Dbusiness&matcher=job%3Dweather-metrics-component&matcher=severity%3Dcritical)  
Dashboard: <http://localhost:3000/d/ZgBkHNhMz?orgId=1>  
Panel: <http://localhost:3000/d/ZgBkHNhMz?orgId=1&viewPanel=6>

**[FIRING:1] Apdex Score Below 0.69 business (business weather-metrics-component critical)**

 Grafana v9.5.2

**Alerta simulada de apdex warning únicamente por el canal Discord**



## Estrategia a nivel infraestructura

Luego, a nivel **infraestructura**, se planteó configurar las alertas primordiales tales como el **liveness probe** de los servicios (utilizando sus **health check endpoints**), sus **latencias**, **circuit breaker**, **cpu**, **storage** y **memoria** de los **containers** y **hosts**. Preferimos priorizar dichas alertas frente a otras porque requerirán un tiempo de maduración y experiencia del equipo para crearlas, o bien mejorarlas a medida que se van probando y experimentando con el sistema a nivel productivo.

**High cpu load**: si algún **nodo** supera la carga de **cpu** promedio de **1.5**, se dispara la alerta.

**High memory load**: si algún **nodo** supera la capacidad de **ram** del **85%**, se dispara la alerta.

**High storage load**: si algún **nodo** supera la capacidad de **disco** o **almacenamiento** del **85%**, se dispara la alerta.

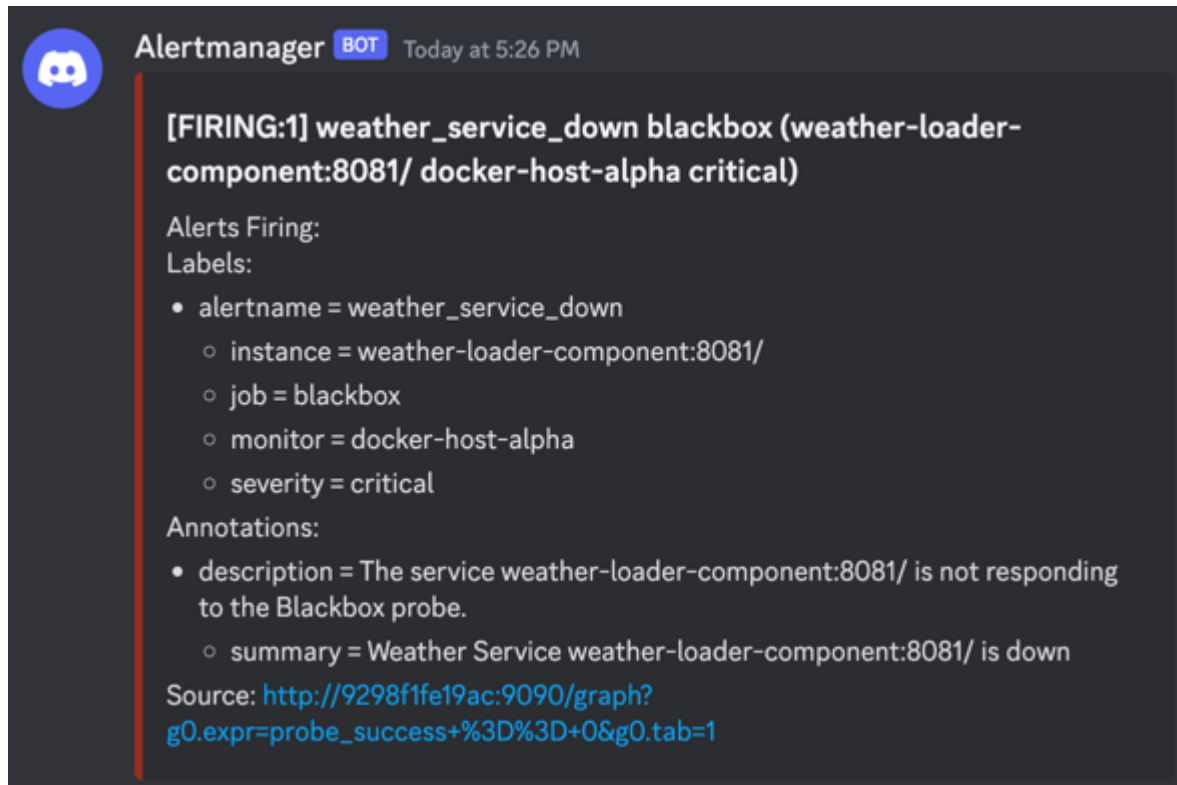
**Monitor service down**: si algún nodo se encuentra caído, se dispara que dicha instancia se encuentra en ese estado.

**Liveness probe (weather service down)**: como se mencionó en otras secciones, se utiliza **blackbox** y esta se dispara cuando algún **endpoint** de health-check **no** responde un http status code entre **[200-299]**.

**Circuit Breaker**: Esta alerta se dispara si el estado del **circuit breaker** en **WeatherMetricsService** se encuentra **abierto**.

## Capturas de alertas simuladas de infraestructura por los canales actuales

### Ejemplo de alerta disparada cuando un servicio no responde al ping



## Ejemplo de alerta y resolución cuando aumenta el consumo del cpu en el host

# alertas\_infra

Source: [http://d3330e2949bb:9090/graph?g0.expr=node\\_load1+%3E+1.5&g0.tab=1](http://d3330e2949bb:9090/graph?g0.expr=node_load1+%3E+1.5&g0.tab=1)



Alertas Infraestructura BOT 29/06/2023 14:35

### [FIRING:1] high\_cpu\_load nodeexporter (infrastructure nodeexporter:9100 docker-host-alpha warning)

Alerts Firing:

Labels:

- alertname = high\_cpu\_load
  - category = infrastructure
  - instance = nodeexporter:9100
  - job = nodeexporter
  - monitor = docker-host-alpha
  - severity = warning

Annotations:

- description = Docker host is under high load, the avg load 1m is at 3.92. Reported by instance nodeexporter:9100 of job nodeexporter.
  - summary = Server under high load

Source: [http://f66f6a1183f4:9090/graph?g0.expr=node\\_load1+%3E+1.5&g0.tab=1](http://f66f6a1183f4:9090/graph?g0.expr=node_load1+%3E+1.5&g0.tab=1)

### [RESOLVED] high\_cpu\_load nodeexporter (infrastructure nodeexporter:9100 docker-host-alpha warning)

Alerts Resolved:

Labels:


- alertname = high\_cpu\_load
  - category = infrastructure
  - instance = nodeexporter:9100
  - job = nodeexporter
  - monitor = docker-host-alpha
  - severity = warning

Annotations:

- description = Docker host is under high load, the avg load 1m is at 1.71. Reported by instance nodeexporter:9100 of job nodeexporter.
  - summary = Server under high load

Source: [http://f66f6a1183f4:9090/graph?g0.expr=node\\_load1+%3E+1.5&g0.tab=1](http://f66f6a1183f4:9090/graph?g0.expr=node_load1+%3E+1.5&g0.tab=1)

## Ejemplo alerta por Circuit Breaker abierto

 Alertmanager BOT Today at 8:05 PM

**[FIRING:1] circuit\_breaker\_open weather-metrics-component (weather-metrics-component:8082 docker-host-alpha critical)**

Alerts Firing:


Labels:

- alertname = circuit\_breaker\_open
  - instance = weather-metrics-component:8082
  - job = weather-metrics-component
  - monitor = docker-host-alpha
  - severity = critical

Annotations:

- description = A Circuit Breaker has been open for more than 10 seconds
  - summary = Circuit Beaker in weather-metrics-component:8082 is open

Source: [http://32e8092e5841:9090/graph?g0.expr=circuit\\_breaker\\_status+%3D%3D+2&g0.tab=1](http://32e8092e5841:9090/graph?g0.expr=circuit_breaker_status+%3D%3D+2&g0.tab=1)

 Alertmanager BOT 06/24/2023 9:00 PM ×

**[RESOLVED] circuit\_breaker\_open weather-metrics-component (weather-metrics-component:8082 docker-host-alpha critical)**

Alerts Resolved:

Labels:

- alertname = circuit\_breaker\_open
  - instance = weather-metrics-component:8082
  - job = weather-metrics-component
  - monitor = docker-host-alpha
  - severity = critical

Annotations:

- description = A Circuit Breaker has been open for more than 10 seconds
  - summary = Circuit Beaker in weather-metrics-component:8082 is open

Source: [http://94796e70dff9:9090/graph?g0.expr=circuit\\_breaker\\_status+%3D%3D+2&g0.tab=1](http://94796e70dff9:9090/graph?g0.expr=circuit_breaker_status+%3D%3D+2&g0.tab=1)

# Tests de carga y dashboards de métricas

En la carpeta **artillery**, se encuentran los archivos para ejecutar los tests de carga. Existen archivos de tests que representan una carga **normal**, **insane**, **super-insane** y **super-mega-insane**. Estos a su vez representan un claro incremento en la masividad de envío de requests y tiempos de estrés.

Es necesario instalar las dependencias junto a **artillery** de forma global para ejecutarlos por comando y que se encuentre levantado el servicio a testear. O bien, se puede descargar la imagen docker “**artilleryio/artillery**” en docker-hub e instanciarlo con visibilidad en la red y conteniendo los archivos de tests correspondientes.

## Local

```
npm install -g artillery
npm install
artillery run artillery/<test-file>.yaml
```

## Docker (dentro del repositorio weather-documentation)

```
docker run --rm -it -v ${PWD}:/repo artilleryio/artillery:latest run
/repo/artillery/<test-file>.yaml
```

Luego de correr los tests de carga, las métricas se ven modificadas y es posible visualizarlas en grafana logs y además los dashboards construidos sobre las métricas.

## Ejecución de intensidad “super-mega-insane” desactivando la caché





En el **primer** gráfico, vemos luego de la línea planchada la corrida **super mega insane** que se eleva y mantiene picos entre los 12k y 15k requests por segundo. Se puede ver el impacto sobre la app en los demás gráficos, por ejemplo, el **segundo** es **Apdex** (promedio de satisfacción respondiendo) teniendo variaciones entre 0.2 (20%) y 0.6 (60%) demostrando un gran estrés en el servidor, manteniendo valores entre medios y bajos de satisfacción.

El **tercer** gráfico junto con el que está debajo representan el porcentaje de requests respondidos por tiempo donde demuestran que el mayor porcentaje de requests se responden en menos de 0.03 segundos pero podemos ver que no llegan ni siquiera a un 40% en total por lo que el resto se pierden o no son respondidos. El **cuarto** gráfico representa el tiempo promedio de respuesta y podemos ver que el promedio varía entre 15 ms y 25 ms con picos cercanos a 30 ms.

## Ejecución de intensidad “super-mega-insane” activando la caché



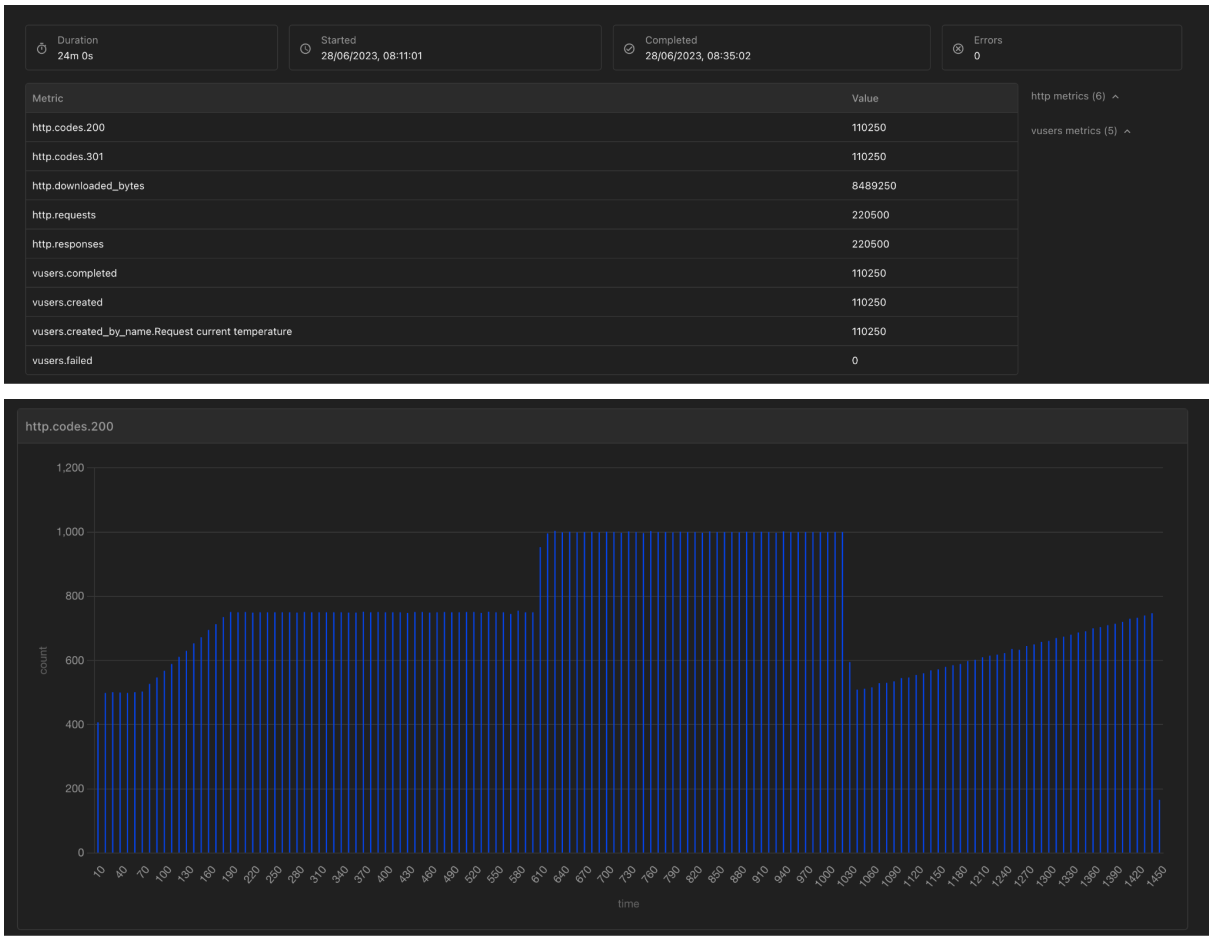
En el **primer** gráfico, vemos luego de la línea planchada la corrida **super mega insane** que se eleva y mantiene picos que no superan los 12.5k requests por segundo. Se puede ver el impacto sobre la app en los demás gráficos, por ejemplo, el **segundo** es **Apdex** (promedio de satisfacción respondiendo) manteniendo una constancia en 1(100%) demostrando un estrés nulo en el servidor y manteniendo valores máximos de satisfacción.

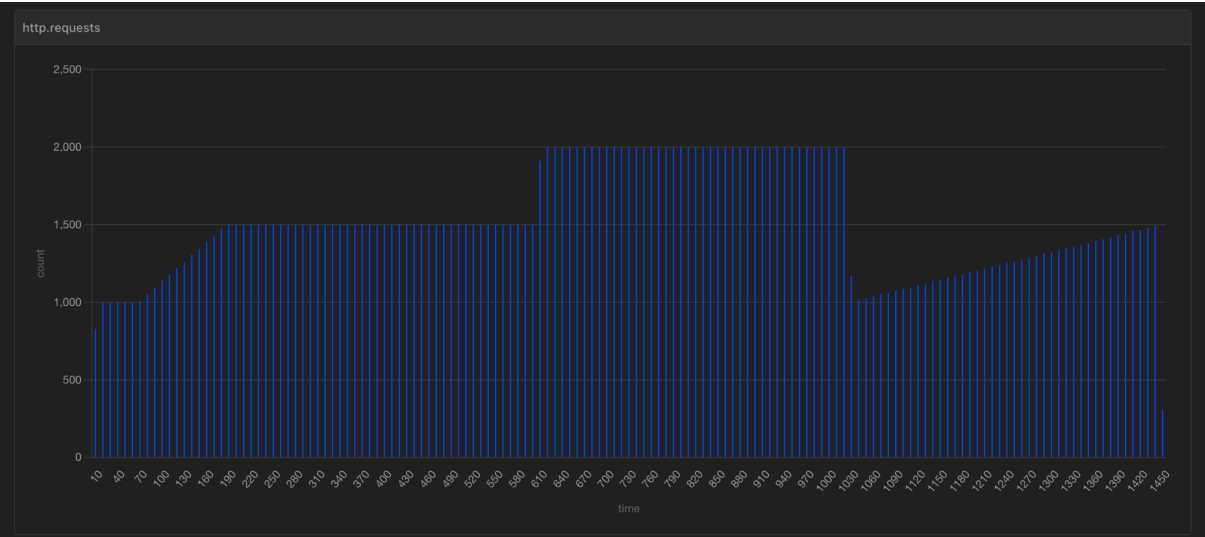
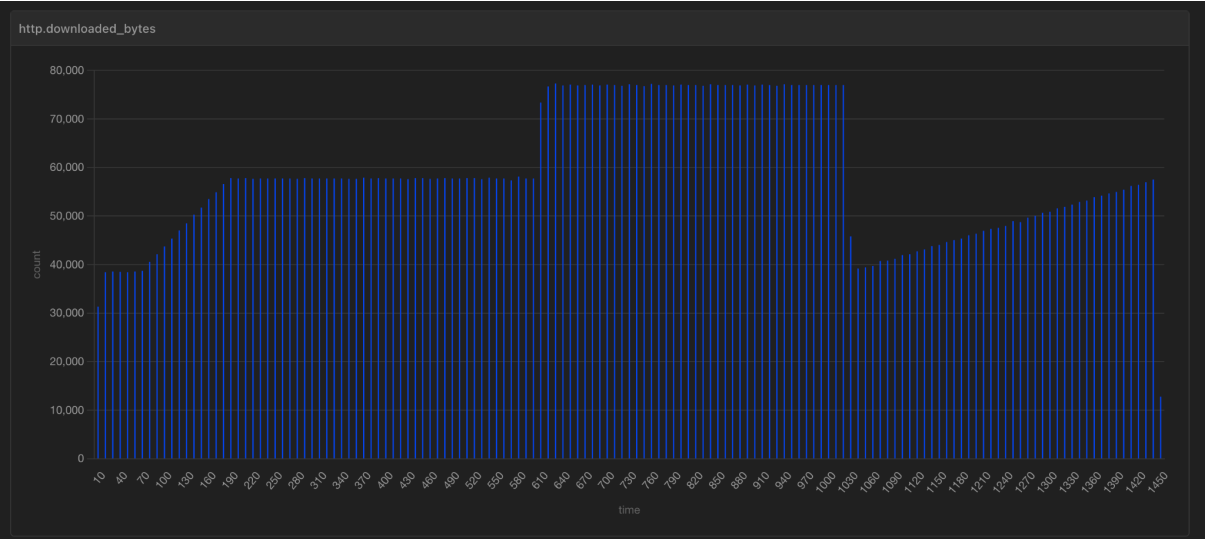
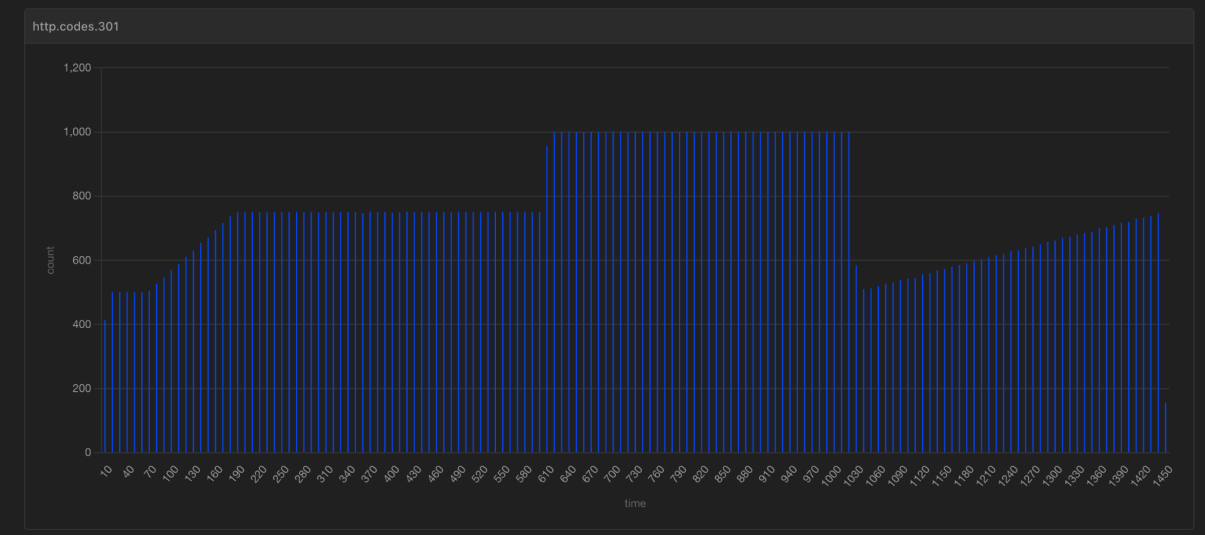
El **tercer** gráfico junto con el que está debajo representan el porcentaje de requests respondidos por tiempo que demuestran que el 100% de los requests fue respondido y en menos de 0.03 segundos cada uno. El **cuarto** gráfico representa el tiempo promedio de respuesta y podemos ver que el promedio varía comienza en 1.3 ms, bajando hasta mantenerse constante en valores de casi 0.

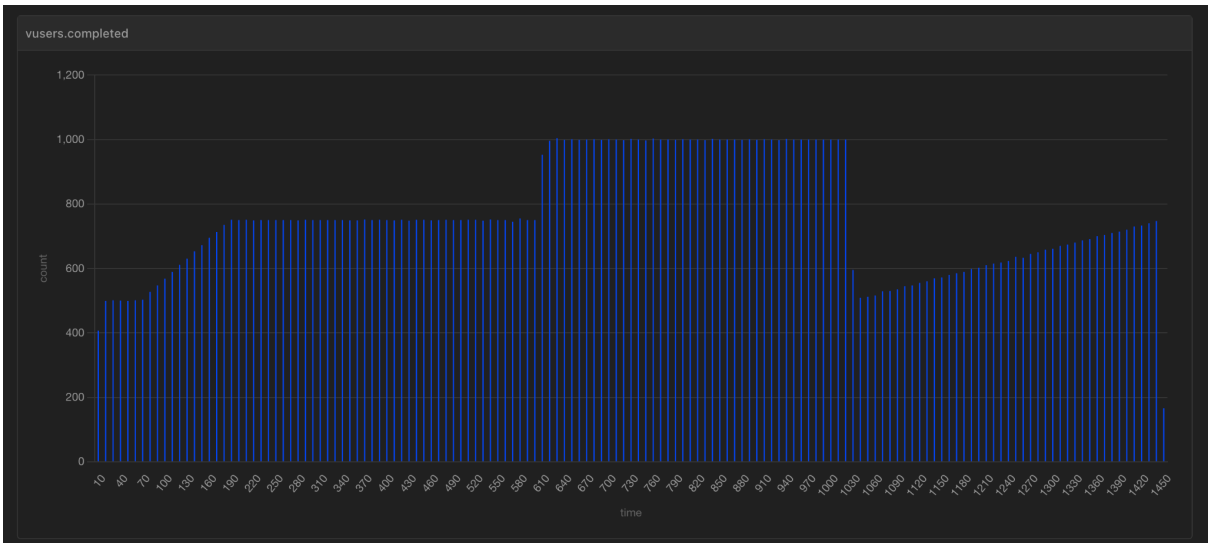
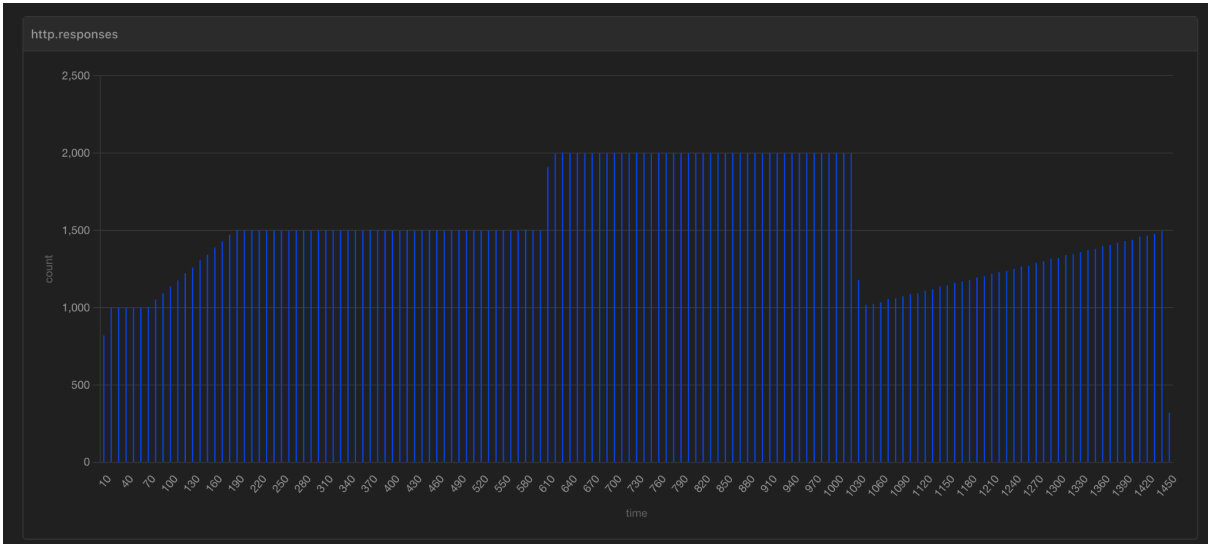
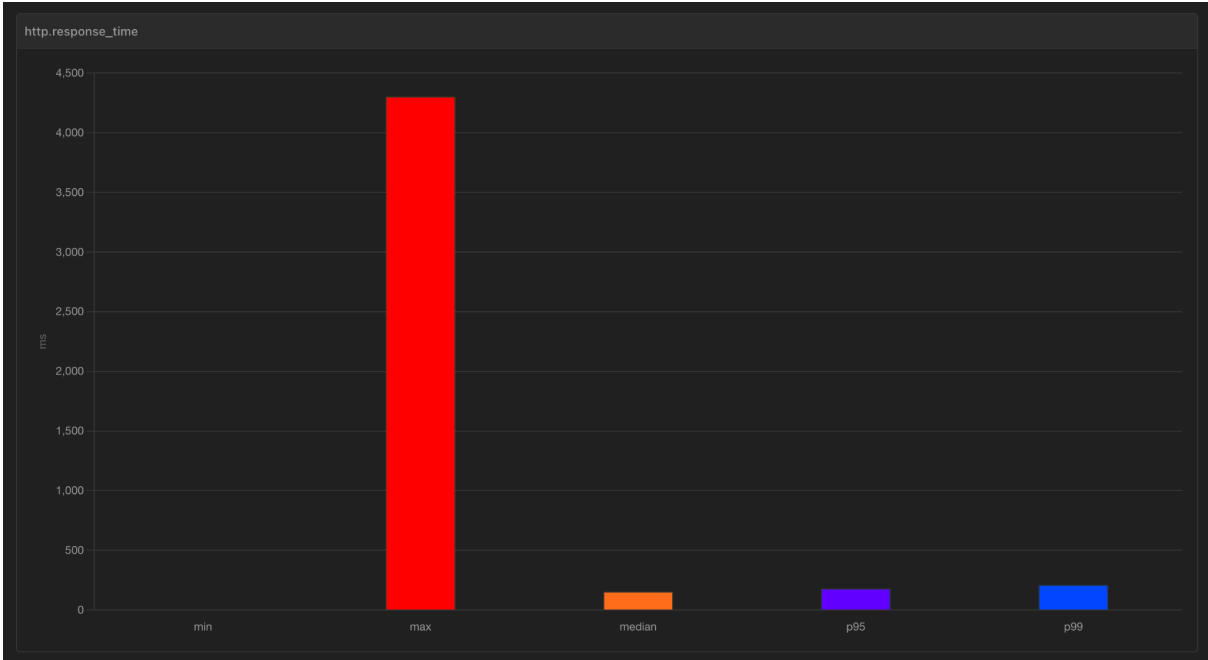
Además, Artillery cuenta con reportes propios que salen en json pero pueden ser convertidos a html para visualizar de mejor forma dicho reporte con los siguientes comandos:

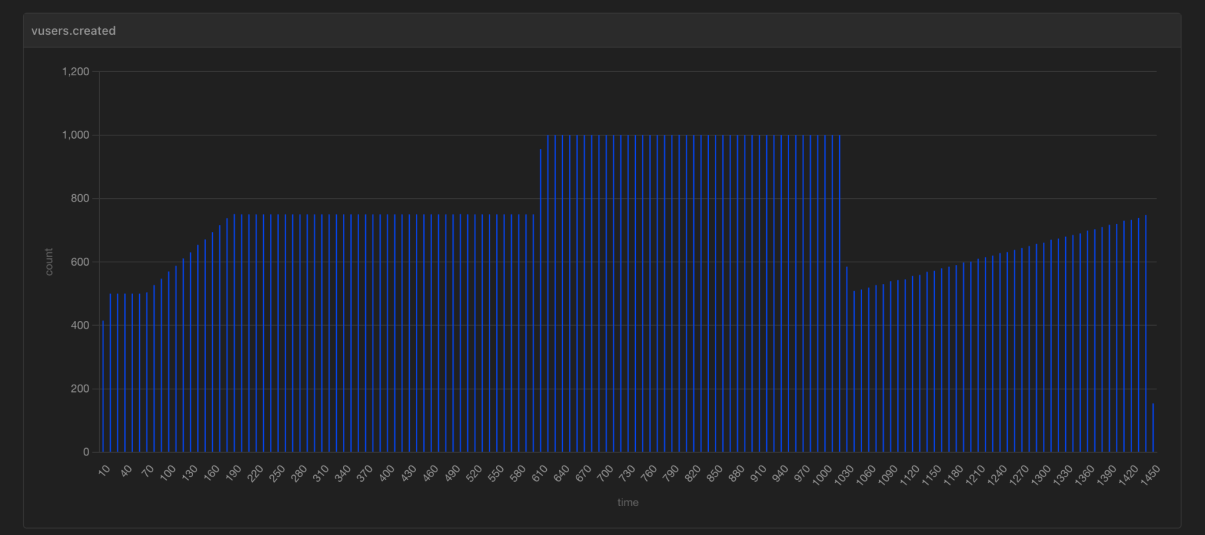
```
artillery run --output report.json artillery/test.yml
artillery report --output report.html report.json
open report.html
```

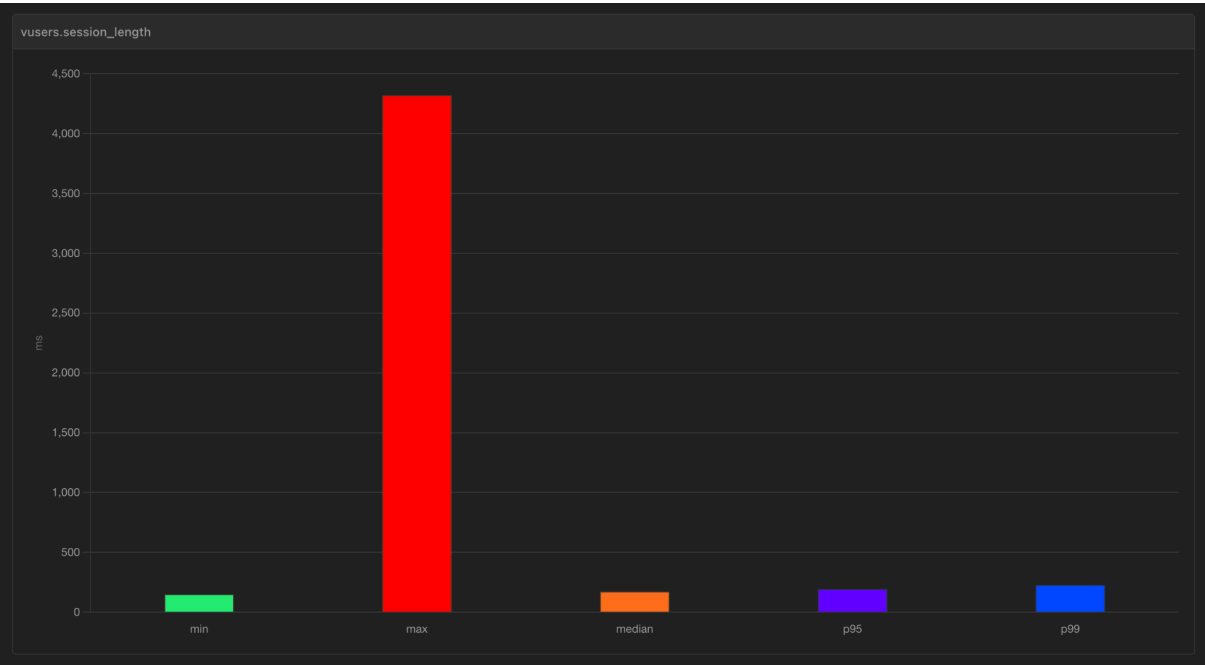
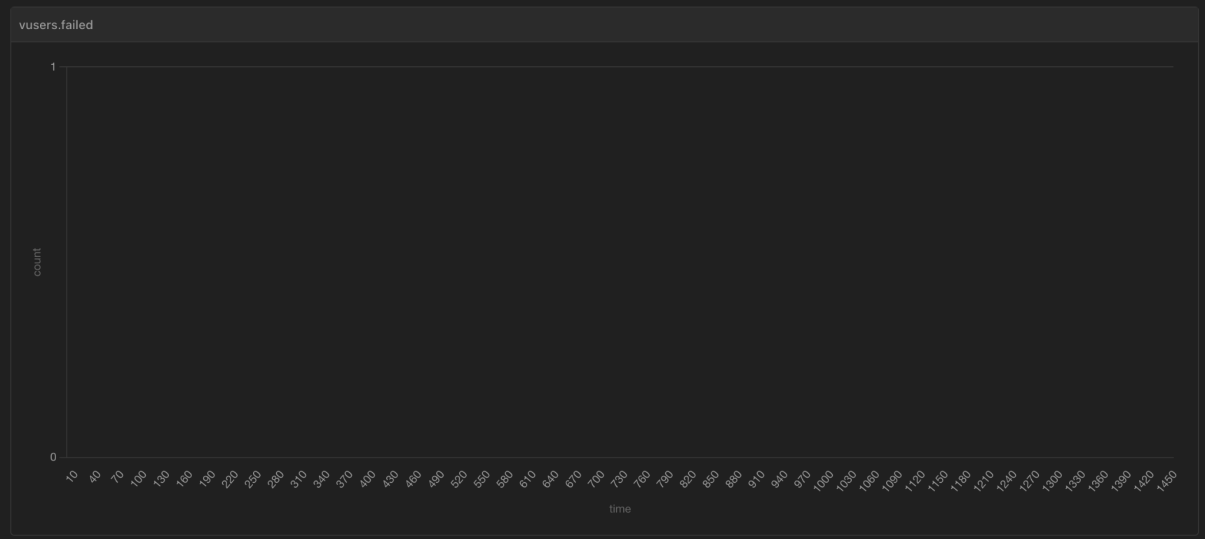
Reporte y gráficos producido por Artillery











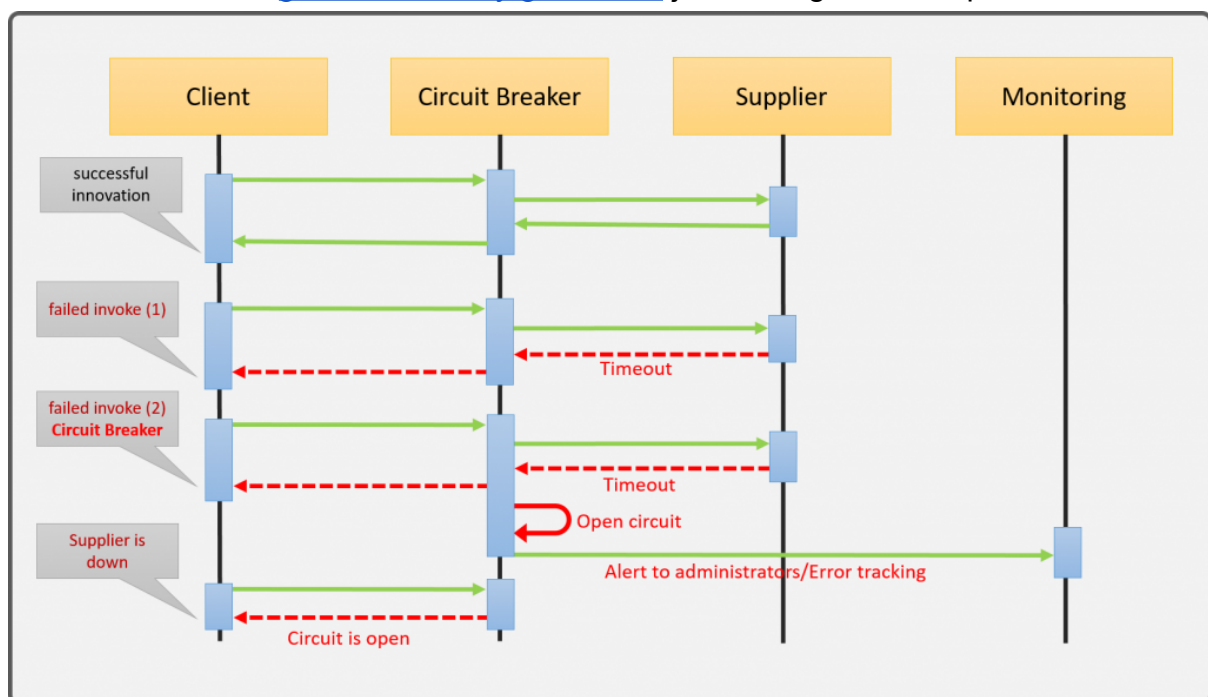
## Time-out & Retries

Los servicios de **Weather Loader** y **Metrics**, poseen en sus variables de entorno la posibilidad de ajustar los tiempos de **timeout** y **retry** con sus configuraciones (*en este ultimo, seria la cantidad de reintentos y la tiempo de espera entre cada reintento*) de los clientes de **http**. Para proveer estas dos características, se utilizaron las siguientes librerías de go [github.com/hashicorp/go-cleanhttp](https://github.com/hashicorp/go-cleanhttp) y [github.com/hashicorp/go-retryablehttp](https://github.com/hashicorp/go-retryablehttp). Una característica buena de estas **libraries**, es que se puede proveer el logger propio para que loguee con el “**system-trace-id**” y cualquier campo provisto propio, para entender cuando inicializa el **retrying**, cada uno de sus intentos con sus errores y el disparador interno del servicio donde se ejecutó.

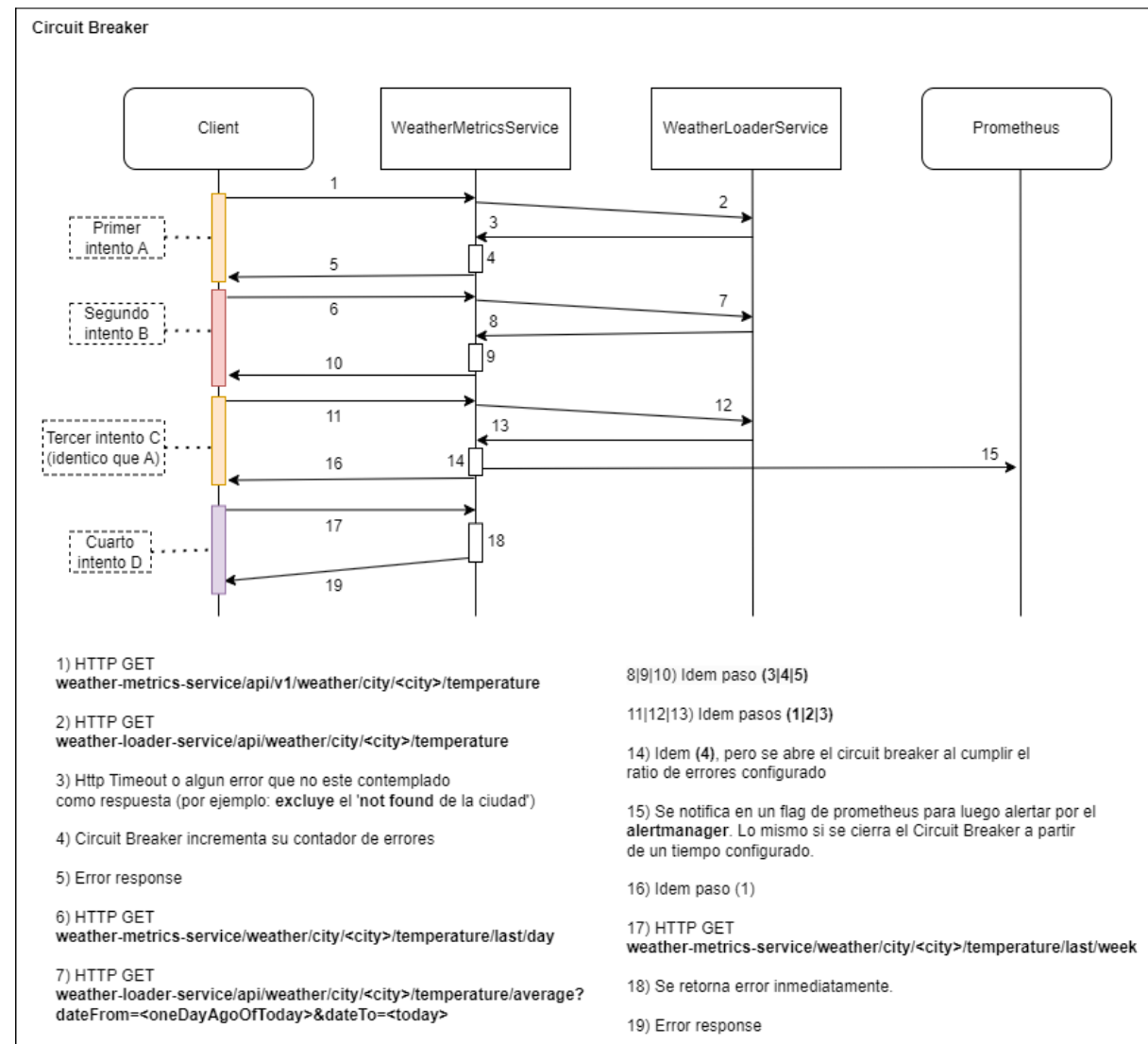
En el caso del **Loader**, además posee un timeout para su cliente de **mongo**. Por otro lado, en el caso del **Metrics** posee un timeout para su cliente de **redis**. Ambos configurables por su library provista por el lenguaje.

## Circuit Breaker

El circuit breaker se agregó en el servicio de **WeatherMetricsService** cuando realiza conexiones contra **WeatherLoaderService**. El ratio de error para la apertura y el tiempo de cierre del Circuit Breaker se encuentran configurables, como primera instancia se configuró con un porcentaje de error del 60%. Para integrar este patrón, se utilizó la librería [github.com/sony/gobreaker](https://github.com/sony/gobreaker) junto al siguiente esquema:



# Diagrama de implementación

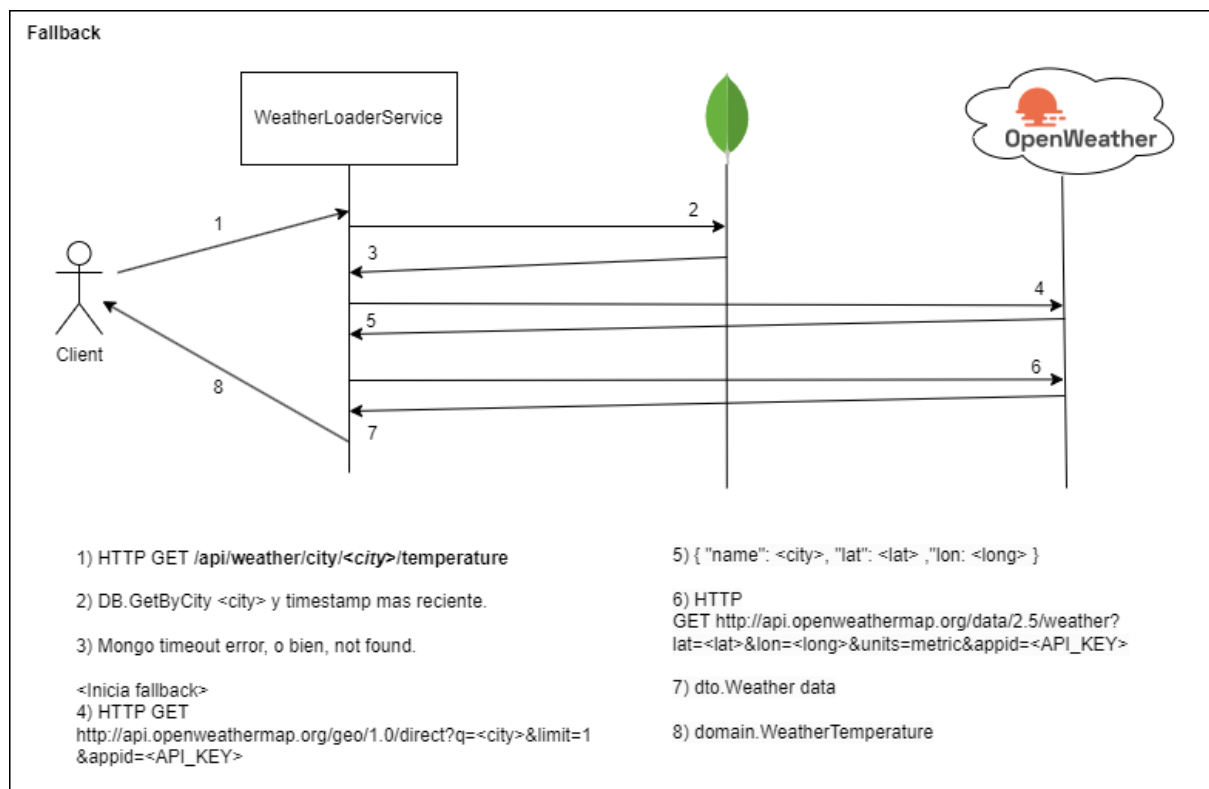




# Fallback

El **fallback** se encuentra en el servicio de **WeatherLoaderService** y se dispara únicamente en el caso de obtener el reporte de la temperatura actual. Esto ocurre cuando el repositorio de Mongo responde algún error (ya sea, un error de servidor, timeout, o bien no encuentra la ciudad solicitada en la **collection**). El servicio lo resuelve yendo a buscar a la API de clima externo. Como es una ciudad lo que se recibe, se realiza una **geo-localización** de la **ciudad**, y luego se va a buscar normalmente a la API con la **geo-coordenada** correspondiente a la ciudad obtenida por la misma API implementada para la obtención de temperaturas actuales.

## Diagrama de implementación



# Request caching

Se optó por realizar el esquema **client-side caching** en **WeatherMetricsService** donde se consumen los recursos del servicio **WeatherLoaderService**, y se utiliza **Time to Live (TTL)** como invalidación de los datos cacheados. Las implementaciones fueron sobre estructuras de datos en memoria **clave-valor**. Con este caching podremos tener una tolerancia a fallos, en caso de que se quiera obtener una ciudad ya cacheada y el servicio de **Loader** esté caído temporalmente.

## TTL

La razón acerca de la implementación de **TTL** como invalidación es debido a la **frecuencia** que es actualizada la información en la base de datos, y en los **cálculos** con fechas en el servicio mencionado. Los recursos que posee dicho servicio son la obtención de la temperatura actual cada hora con un timestamp generado al persistirlo, y la obtención de la temperatura promedio entre dos fechas.

En el primer caso, el cálculo del tiempo de **expiración** es la diferencia entre el **timestamp** que posee la última medición persistida sumado con una hora posterior, menos el mismo **timestamp**.

En el otro caso, la consulta de promedios de fechas del lado del **cliente** siempre son con ambas fechas a las "00:00" GMT-3 ó "03:00" UTC con el rango de un ó siete día/s anterior/es al de hoy junto a la fecha de hoy. Además, el **promedio** de temperatura de un día siempre va a ser el promedio entre su temperatura **mínima** y su **máxima**. Con estas premisas, el cálculo del tiempo de **expiración** para este caso sería el tiempo que falta para que finalice el día de hoy que se realiza la consulta.

## Implementación

Se realizaron pruebas con dos implementaciones de las cuales, una fue a nivel local y otra a nivel externo para concluir con la que sea más performante, resiliente y escalable.

En una primera instancia, se realizó la implementación con una librería ([github.com/patrickmn/go-cache](https://github.com/patrickmn/go-cache)) que por detrás posee un **hashmap** concurrente y thread-safe para un único host, así se evitan llamadas a la red. Al realizar **pruebas de carga** con numerosos requests, se detectaron varios errores a la hora de consumir concurrentemente la presente caché haciendo que el servicio responda con error en algunos casos. Por este motivo y al ser poco **escalable**, se decidió **documentar** y **descartar** esta implementación para evitar el **bug fix** y su **refactor**. Luego, procedimos con la implementación de una caché externa.

En el caso de la caché externa, se implementó con Redis (<https://redis.io/>) con la flexibilidad de poder ser utilizado por varias instancias de un mismo servicio y, también, otros servicios que consuman los recursos de **WeatherLoaderService**.

## Clave/Key

En ambos casos, se utilizó la misma semántica para definir la clave o key para la persistencia de un objeto. La clave se va a componer de un prefijo y sufijo, donde el prefijo se va a diferenciar del recurso obtenido de **WeatherLoaderService** y el sufijo va a depender de la ciudad en ambos casos y del rango de las fechas consultadas separadas por “\_” en el caso del promedio únicamente. Ejemplos:

- Prefijos: “weather:current” ó “weather:average”.
- Sufijos: “Quilmes” ó  
“Quilmes\_2023-06-16T03:00:00Z\_2023-06-17T03:00:00Z”.

# Adicional: Dashboard de temperaturas

Quisimos sumar la data del tiempo como métricas para poder visualizarlas en Grafana, pero tuvimos algunas limitaciones.

Al utilizar docker-compose local y no tener hosteado el sistema en la nube, nuestra única DB persistente en el tiempo es MongoDB mediante MongoAtlas.

**Opción 1:** Sumar métricas tradicionales de “pull” a Prometheus no iba a funcionar, porque solo íbamos a tener datos de temperaturas de los últimos minutos.

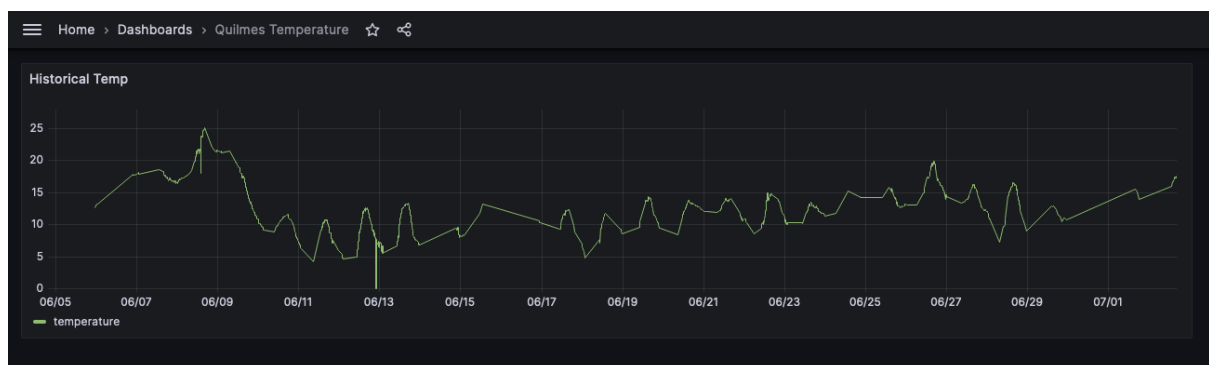
**Opción 2:** El “backfilling” de métricas en Prometheus por el momento no está solucionado de manera oficial, por lo que tampoco fue una opción.

**Opción 3:** Agregar a MongoDB como datasource de Grafana y poder explotar la información de las temperaturas. Lamentablemente el plugin solo está disponible para la versión Enterprise.

**Opción 3(b):** Usar un plugin “community”, por ej. [este](#).

Llevó un poco más de tiempo instalarlo ya que no está disponible como plugin oficial. De todos modos, no funcionó del todo bien. Era poco intuitivo de usar ya que la UI en Grafana era muy precaria, y tampoco se lo pudo usar con el modo “timeseries” porque a algún elemento de la DB estaba sin timestamp.

**Opción 4 (estática):** Se pudo importar la DB con un plugin libre de CSV en Grafana, y hacer uso de todos los datos, pero al ser un dataset estático y que necesitaba una carga manual decidimos no incorporarlo a la versión final.



# Bibliografía

<https://www.docker.com/>  
<https://docs.docker.com/compose/>  
<https://go.dev/>  
<https://www.mongodb.com/>  
<https://www.oscarblancarteblog.com/2018/12/04/circuit-breaker-pattern/>  
<https://prometheus.io/docs/>  
<https://grafana.com/docs/>  
<https://github.com/sony/gobreaker>  
<https://www.youtube.com/watch?v=hyasWpxP32c>  
[github.com/patrickmn/go-cache](https://github.com/patrickmn/go-cache)  
<https://redis.io/>  
<https://grafana.com/oss/loki/>  
<https://grafana.com/docs/loki/latest/clients/promtail/>  
[https://github.com/prometheus/node\\_exporter](https://github.com/prometheus/node_exporter)  
[https://github.com/oliver006/redis\\_exporter](https://github.com/oliver006/redis_exporter)  
<https://opentelemetry.io/docs/>  
<https://grafana.com/oss/tempo/>  
[https://github.com/prometheus/blackbox\\_exporter](https://github.com/prometheus/blackbox_exporter)  
<https://github.com/prometheus/alertmanager>  
<https://github.com/google/cadvisor>  
<https://www.artillery.io/docs>  
<https://dev.to/saniadsouza/artillery-quick-check-your-site-s-performance-kbf>  
<https://www.artillery.io/docs/docker>  
<https://dev.to/stepanvrany/part-iv-telegram-notifications-e10>  
[https://medium.com/@tristan\\_96324/prometheus-apdex-alerting-d17a065e39d0](https://medium.com/@tristan_96324/prometheus-apdex-alerting-d17a065e39d0)  
<https://www.robustperception.io/sending-alert-notifications-to-multiple-destinations/>  
<https://velenux.wordpress.com/2022/09/12/how-to-configure-prometheus-alertmanager-to-send-alerts-to-telegram/>  
<https://medium.com/devops-dudes/prometheus-alerting-with-alertmanager-e1bbba8e6a8e>  
<https://prometheus.io/docs/alerting/latest/configuration/#matcher>  
<https://grafana.com/docs/grafana/latest/alerting/>