Trabajo Práctico Final:
Documentación

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 cloud o enterprise.

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. Tambien, por su facilidad de tener queries performantes y flexibles.
- **Redis**: Escalable, alta disponibilidad y performante con su sistema de replicación, es opensource, . Tambien, por permitir tener estructuras de datos flexibles.

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 tambien, se complementan entre sí.
Volviendo al primer punto, esto es porque son desarrollados por ellos mismos para ofrecer estas funcionalidades adicionales a las originales.

- **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. 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, Webhooks, 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“**.

Diagramas de Secuencia

Diagrama de temp actual:
<<Diagrama en draw.io>>

Diagrama de temperatura del ultimo dia (o por rango):
<<Diagrama en draw.io>>

Diagrama de arquitectura

<<Diagrama en draw.io>>

Estrategias de Observabilidad

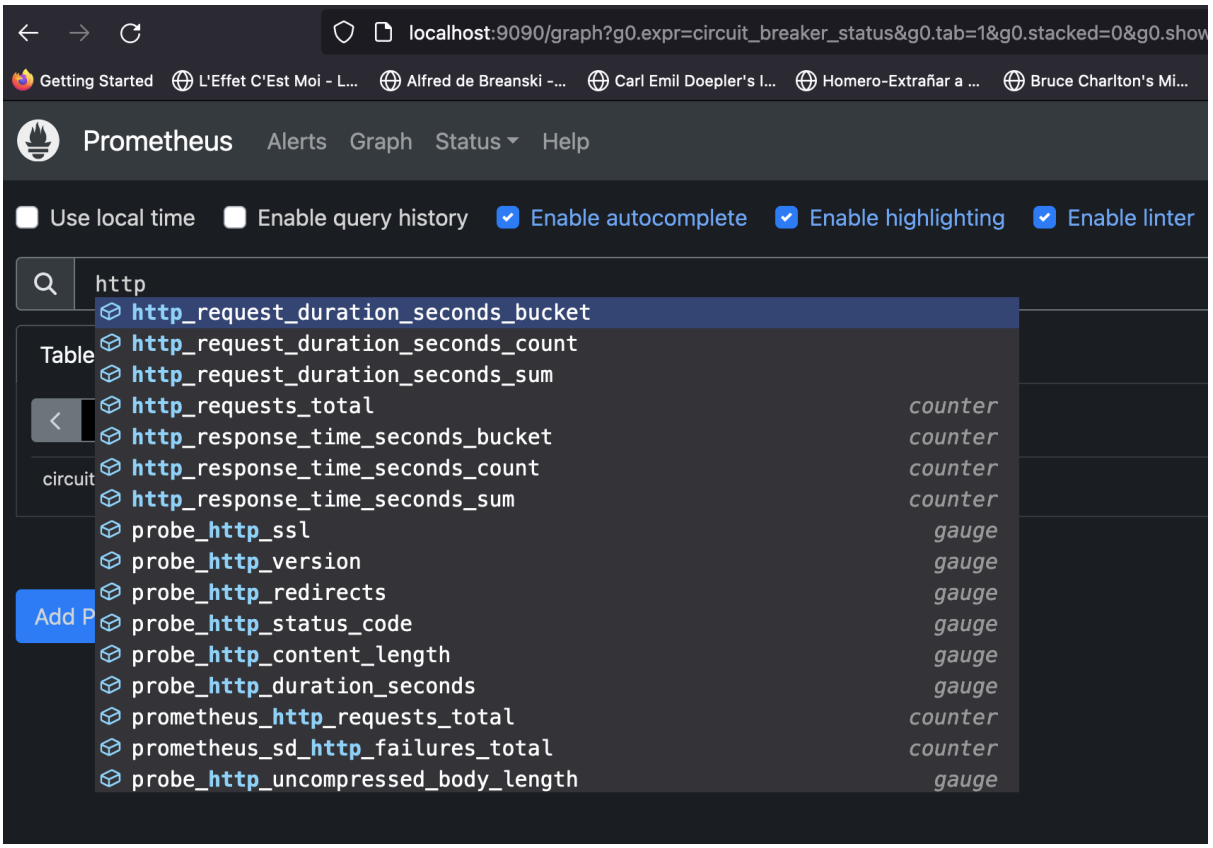
Las estrategias se implementaron fundamentalmente en base con **Prometheus** y **Grafana**. Las tecnologías o herramientas adicionales giran en torno a estas dos mencionadas. Si se requiere más información o enlaces de alguna tecnología a la presente en esta sección, dirigirse a la sección de **tecnologías**.

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

Ejemplo UI:



Ejemplo de scrape configs:

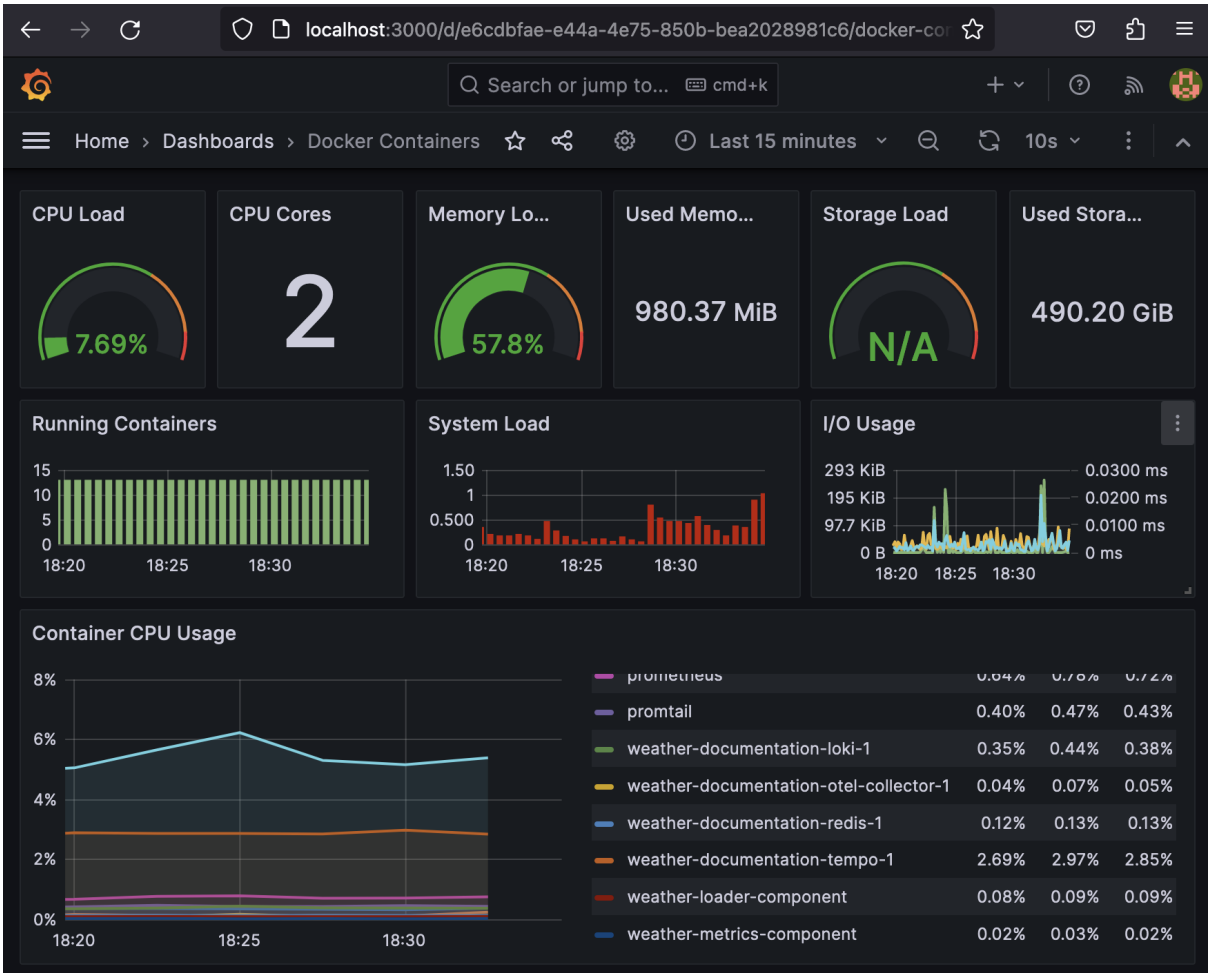
```
# A scrape configuration containing exactly one endpoint to scrape.
scrape_configs:
 - job_name: "nodeexporter"
   scrape_interval: 5s
   static_configs:
     - targets: ["nodeexporter:9100"]

 - 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.

Ejemplo 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"
```

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 4 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".

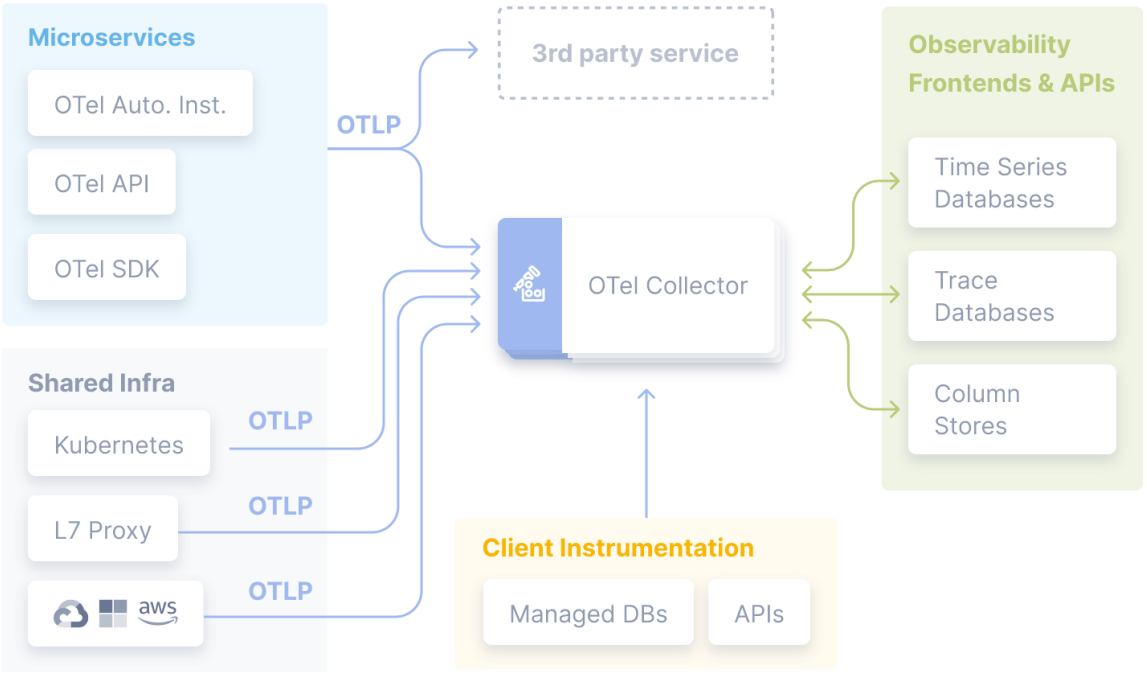
Esas métricas son usadas manualmente para sacar alguna estadística y además se usan para dashboards armados en Grafana para obtener información mucho más importante.

Distributed tracing

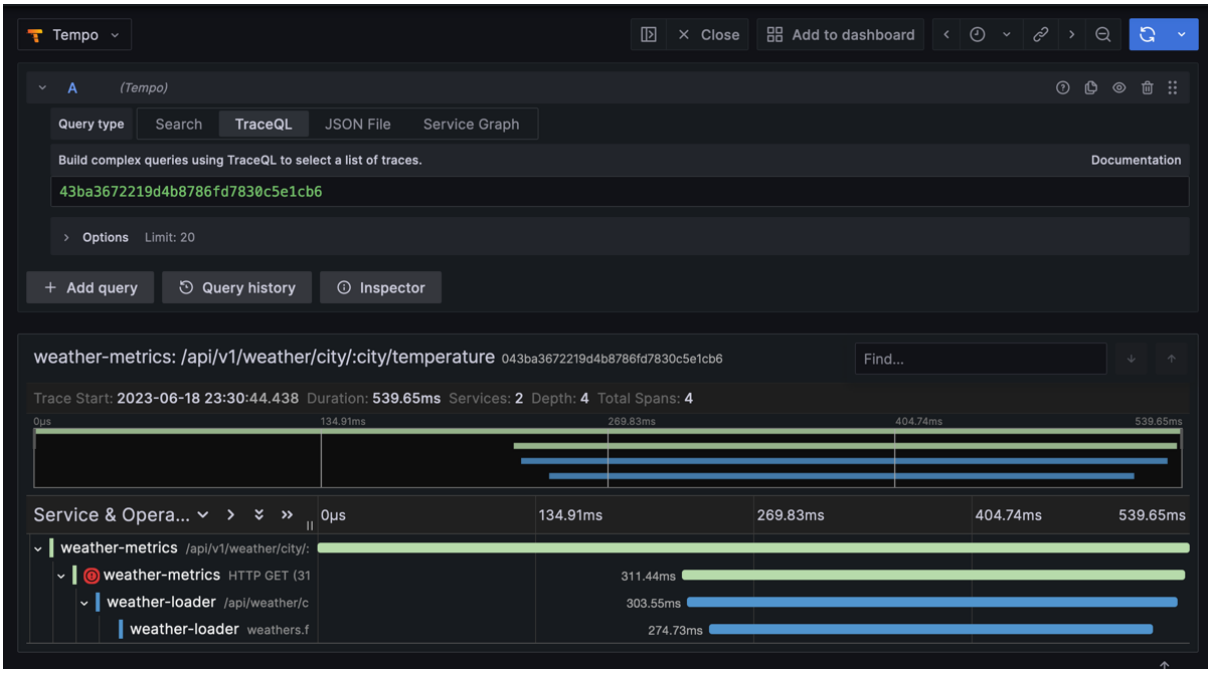
Por un lado, se posee un traceId a nivel código para identificar las transacciones a nivel negocio.
También, se integró **OTel** (Open Telemetry) para obtener una información más exhaustiva y detallada del ciclo de vida de un requests con sus interacciones a nivel de red (clientes de redis, http y mongo).

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**.

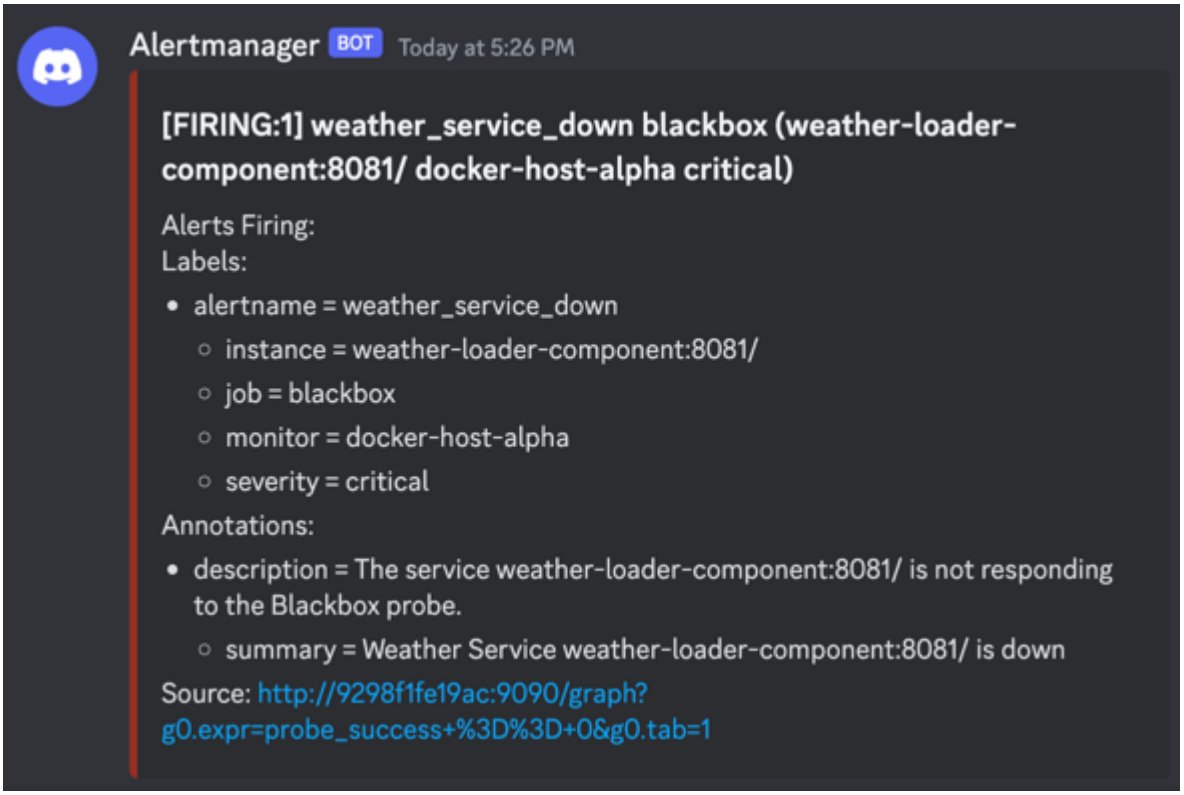


Alerting

Implementado con **Prometheus**, Blackbox Exporter y Alertmanager.

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).

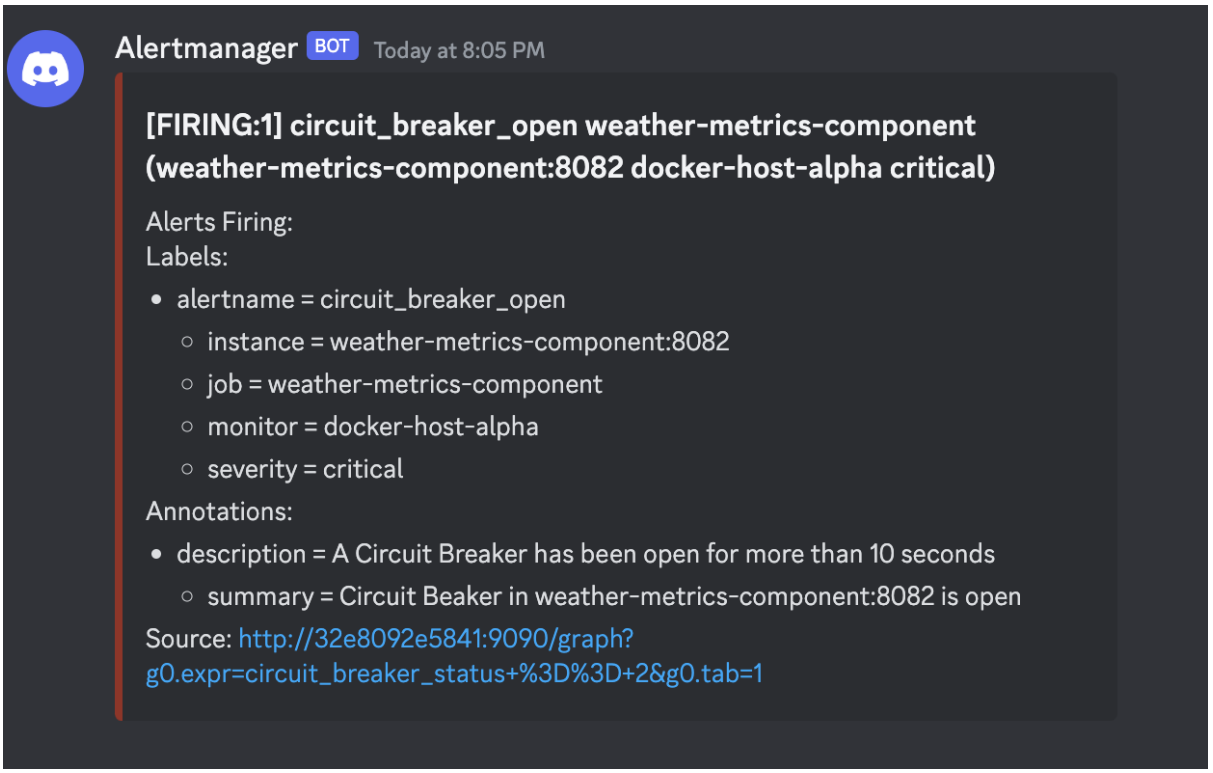
Configuramos un "**scrape**" de **Blackbox** en **Prometheus** para que sondee los servicios del Tiempo (hacer un GET de healthcheck). En caso de no responder, se dispara una alerta en **Prometheus**. La alerta llega a **Prometheus** y es administrada por **alert-manager**, que tiene configurado un **webhook** 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.



Estrategia de alerting

Como estrategia nos planteamos configurar las alertas primordiales tales como el health check de los servicios, sus latencias y circuit breaker??? , etc. Preferimos priorizar dichas alertas frente a otras que requerirán un tiempo de maduración y experiencia del equipo para mejorarlas a medida que se van probando y experimentando.
Las alertas nos llegan a un channel de discord y ahí podremos escucharlas, esta configuración está provista en el alert-manager, y se pueden agregar más medios de alerta a medida que sea necesario.

Ejemplo alerta por Circuit Breaker abierto:



Tests de carga

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>.yml
```

Docker (*dentro del repositorio weather-documentation*):

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

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.

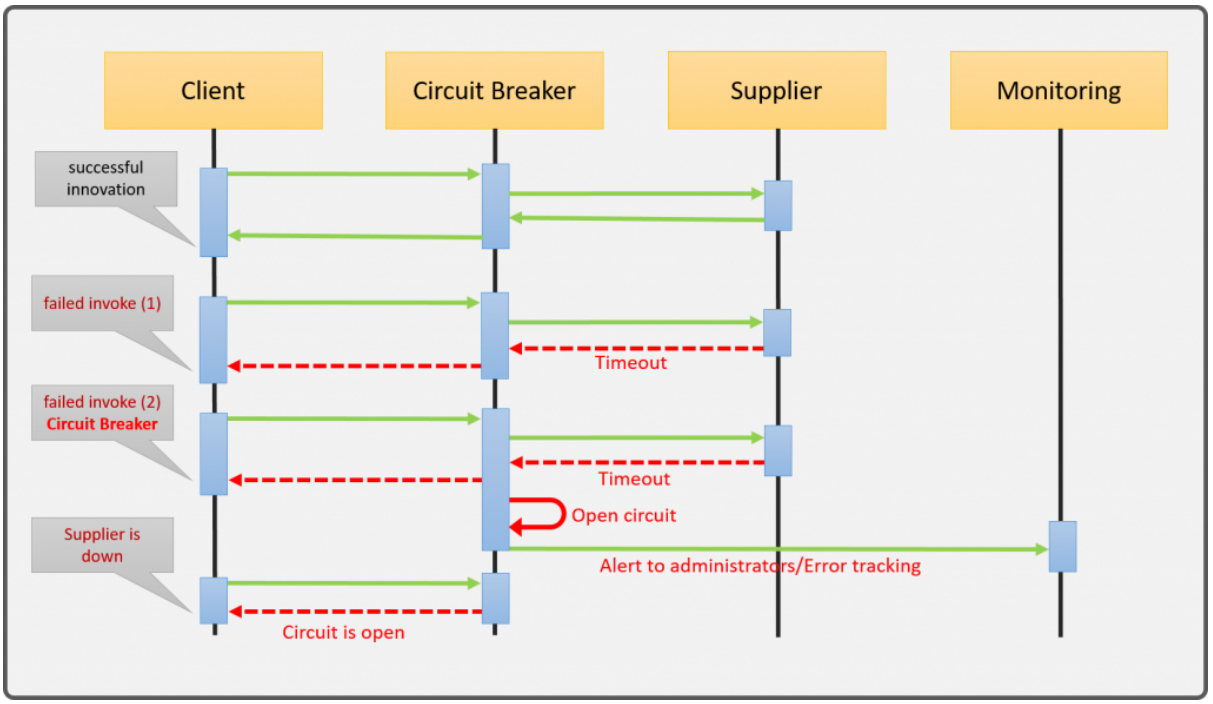
Time-out & Retries

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 librerias de go github.com/hashicorp/go-cleanhttp y github.com/hashicorp/go-retryablehttp.

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 librería provista.

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. Para integrar este patrón, se utilizó la librería github.com/sony/gobreaker junto al siguiente esquema:



Implementación:
<<Diagrama en draw.io>>

Fallback

El **fallback** se encuentra en el servicio Loader cuando se cae Mongo. 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 conseguidas de la misma API.

<<Diagrama en draw.io>>

Caching

Se optó por realizar el esquema **client-side caching** en los recursos del servicio **WeatherLoaderService**, y se utiliza **Time to Line** (**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) 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, tambien, otros servicios que consuman los recursos de **WeatherLoaderService**.

Clave o 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”**.

Bibliografía

https://www.oscarblancarteblog.com/2018/12/04/circuit-breaker-pattern/
https://prometheus.io/docs/
https://grafana.com/docs/
https://github.com/sony/gobreaker