75.61 Taller de Programación III

TP1: Reporte de Pruebas de Carga

Sitio Institucional (aka: Page Visits Counter)

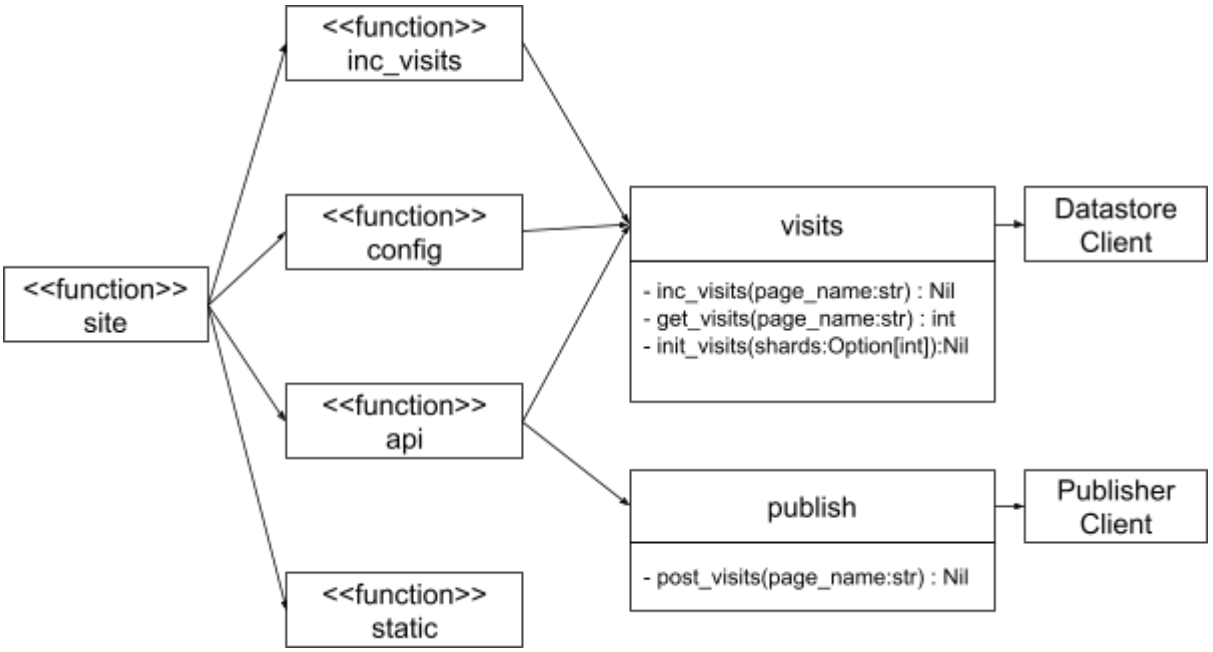
Alumno: 84840 Paulo César Cuneo

1 Arquitectura

Para la arquitectura planteada, tomamos como supuesto que el contador de visitas no tiene requerimientos de tiempo real, y por lo tanto a nivel negocio alcanza que sea correcto bajo consistencia eventual(hay requerimientos sobre el tiempo de convergencia al valor correcto).

1.1 Vista Lógica

1.1.1 Clases



La estructura del programa es bastante simple ya que está diseñado para desplegarse como funciones en Google Cloud Functions.

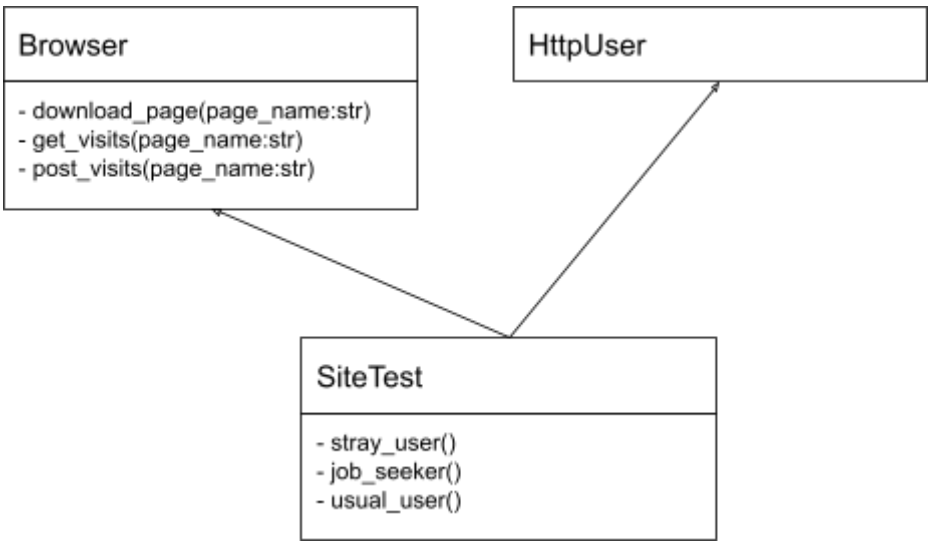
site, config, api, static son funciones que reciben request(se omite en el diagrama)
inc_visits es un functions que recibe event y context(se omiten en el diagrama)

site existe para realizar pruebas locales y es un composite de las demás funciones.

visits y publish en realidad son packages de python; utilizan los clientes de Google Pub/Sub y Google Datastore correspondientemente.

Para las optimizaciones de contadores shardeado y conteo en batch simplemente se modifica la implementación **visits.inc_visits** y **visits.get_visits**

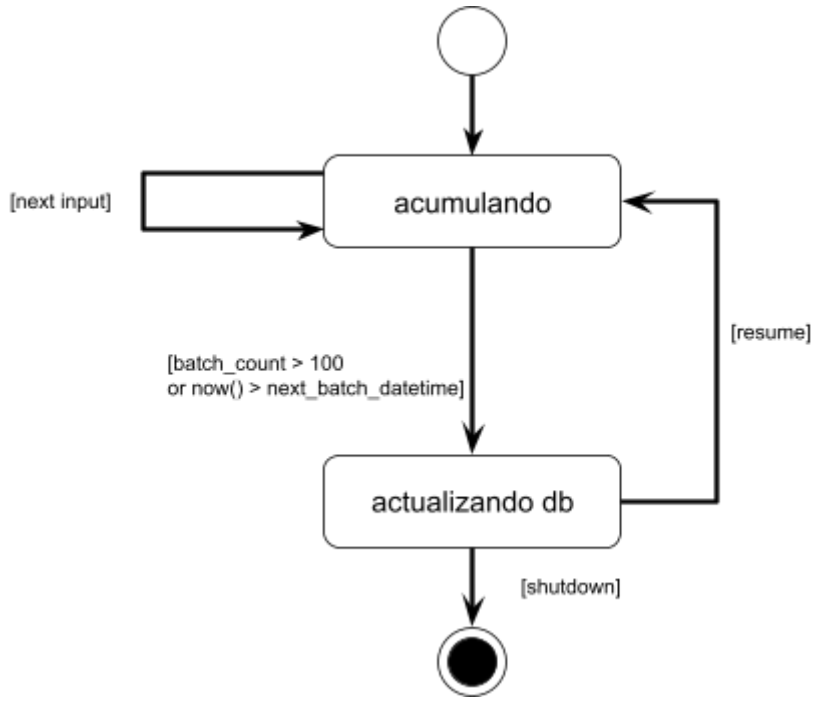
Además para las pruebas de locust utilizamos las siguientes clases.



La clase **Browser** implementa métodos para descargar las páginas con todos los recursos y realizar las peticiones ajax.
SiteTest por su parte reutiliza comportamiento de **Browser** mediante herencia, y por otro lado hereda de **HttpUser** para ser reconocida por locust.io como el punto de entrada.

1.1.2 Estados

En el caso del contador sin sharding, el request simplemente fluye a través del callpath, en cambio para el caso del contador utilizando procesamiento batch tenemos el siguiente diagrama de estados.



En este caso para cada nueva entrada, es decir, request de actualizar visita, se actualiza el contador en memoria hasta alcanzar el batch_count de 100 o que sea la hora de actualizar la base, lo que ocurra primero.

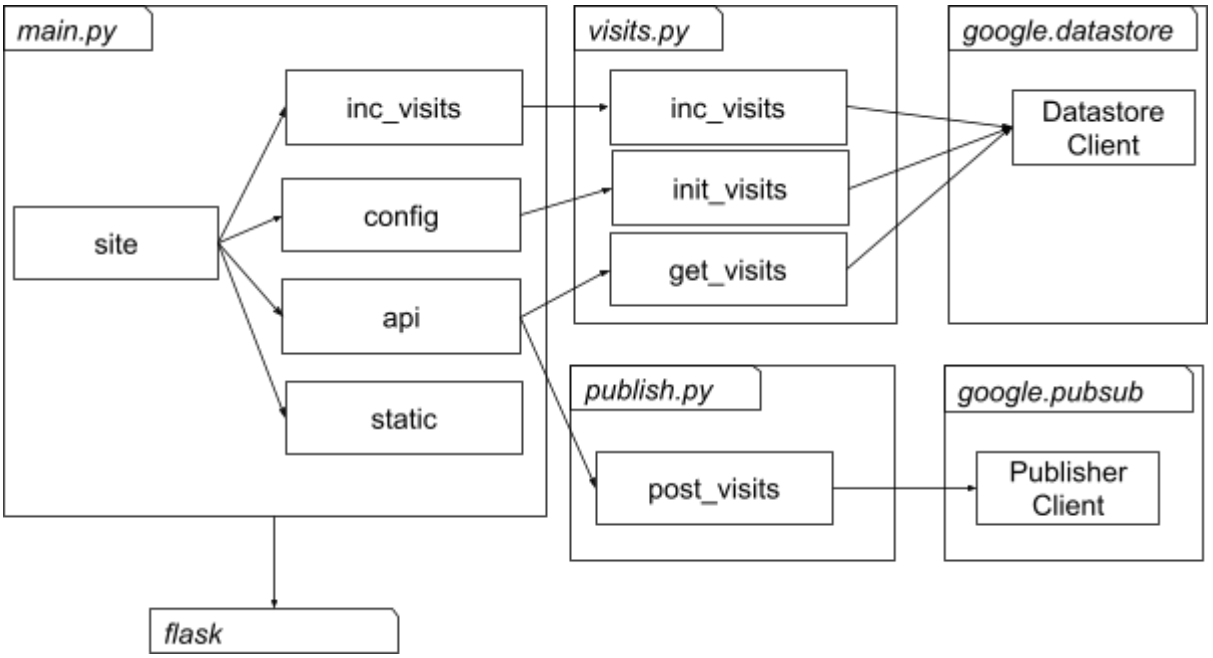
Vale aclarar que, al salir del estado **actualizando db** los contadores quedan actualizados en la base de datos y el contador interno que acumula los cambios se re-inicializa y se actualiza el next_batch_datetime en 10 segundos más que el instante actual.

Tech Debt Note: Por otro lado al funcionar el batch de forma continua y accumular en memoria no es tolerante a fallos, y además el estímulo para transicionar es generado por los request, por lo que el contador puede quedar desactualizado hasta que se reciba un nuevo request.

1.2 Vista Desarrollo

1.2.2 Paquetes

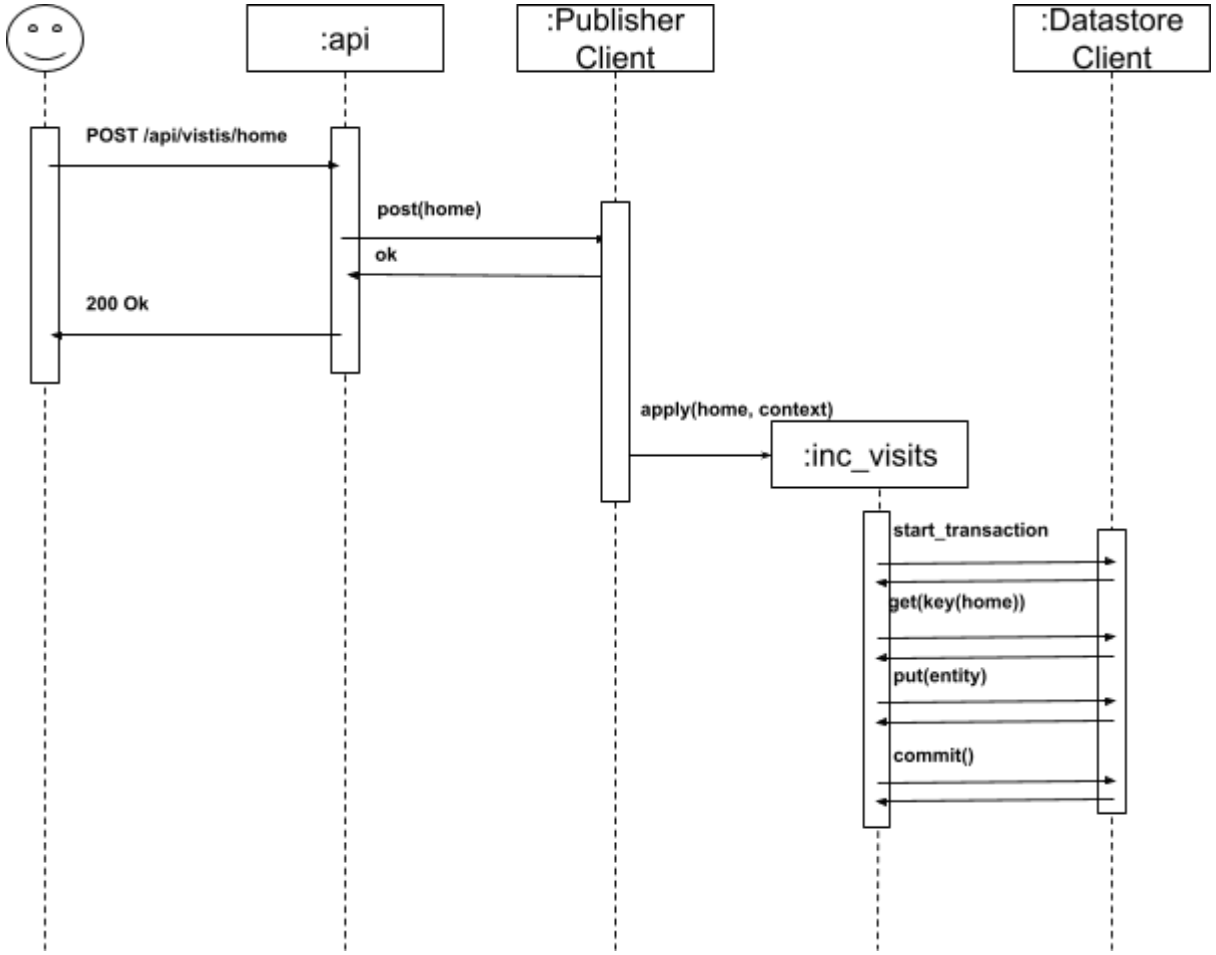
Dada la simplicidad del sistema, existen 3 paquetes principales, **main.py** que contiene los puntos de entrada utilizados por las cloud functions, **visits.py** que adapta el cliente de datastore de google, y **publish.py** que adapta el cliente de pubsub.



1.3 Vista Procesos

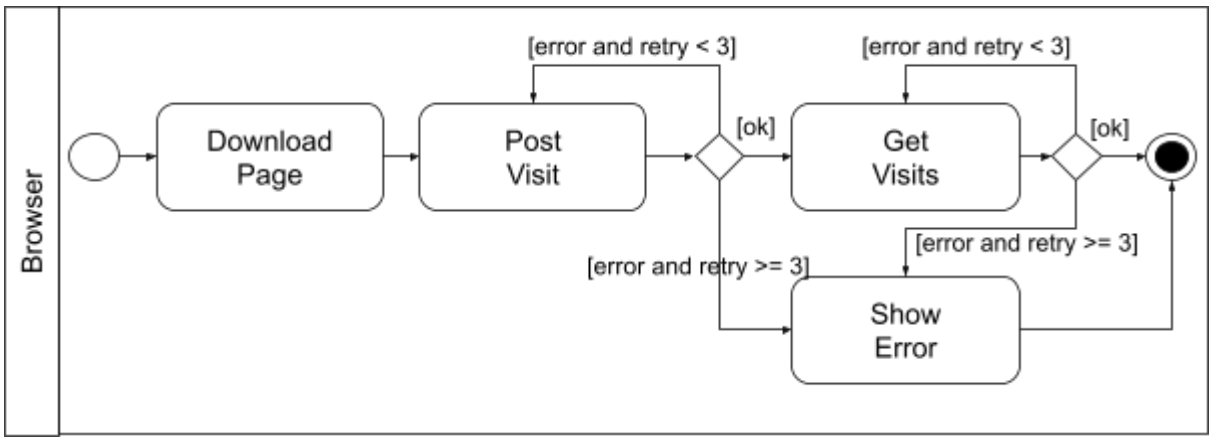
1.3.1 Secuencia

En el siguiente escenario del post, estamos omitiendo el middleware de google y asumiendo que los objetos se comunican directamente.



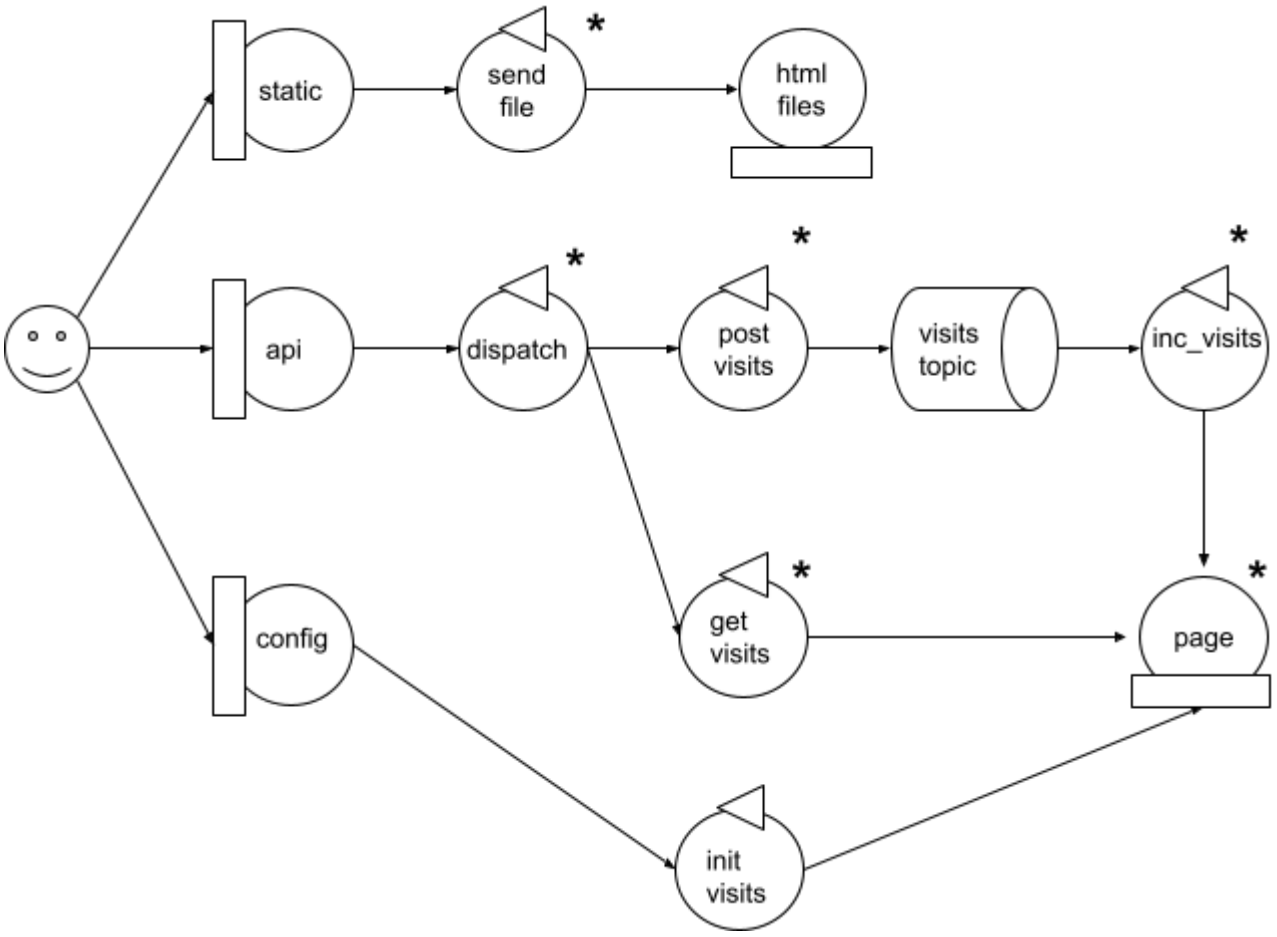
1.3.2 Actividad

Desde el punto de vista del usuario existe una única actividad, la de visitar la página, pero desde el punto de vista del browser existen las siguientes 3 actividades.



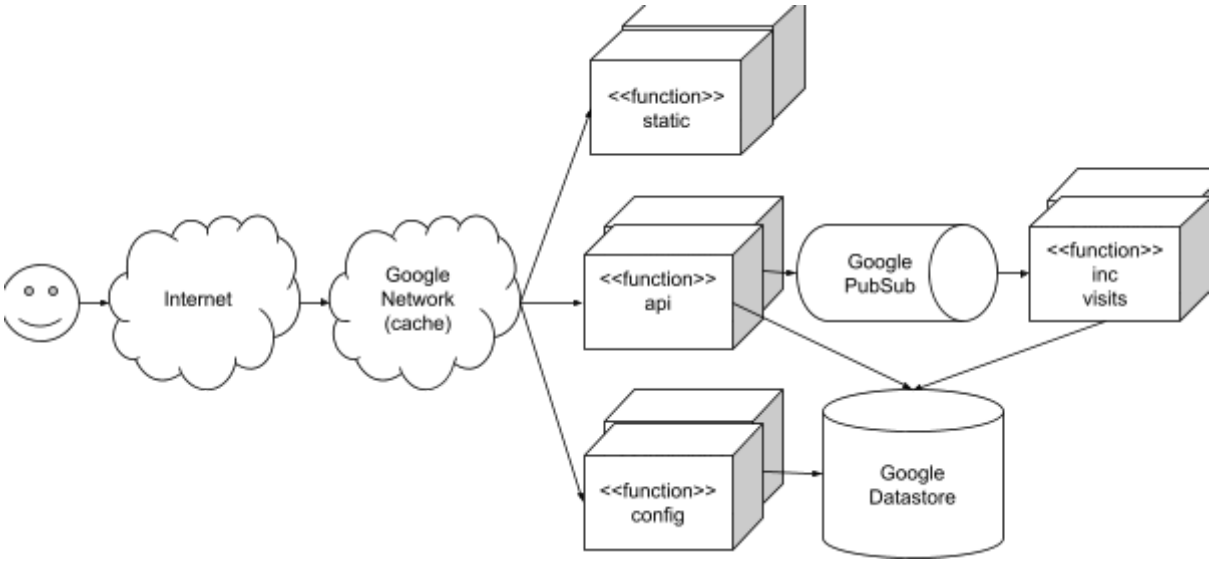
1.4 Vista Física

1.4.2 Robustez



Como se verá en el diagrama de despliegue, el controlador de **dispatch** es desplegado junto con **post_visits** y **get_visits**. Por lo que en la práctica escalan como una unidad.

1.4.1 Despliegue



Tener en cuenta que la red de google posee cache que no son controlados manualmente. En el caso de servir archivos estáticos, esta cache sirve para optimizar la cantidad de descargas de los mismos.

2 Reporte de Pruebas de Performance

2.1 Escenarios de Prueba

Asumimos que existen los siguientes perfiles de usuarios con los respectivos comportamientos.

	Usuario Perdido	Usuario Trabajador	Usuario Habitual
Descripción	Entro al sitio por error, revisa la información del sitio y se va.	Entra al sitio con el interés de ser contratado por la compañía.	Conoce el sitio y sabe cómo usarlo.
Patrón de visita	/home /about	/home /jobs /about /offices /about /home	/home /about /legals /about /home
Ocurrencia	4,8 %	47,6%	47,6%

Para todas las pruebas ejecutadas estos perfiles serán distribuidos de acuerdo a la ocurrencia.

Para cada sección del sitio se simula el comportamiento del usuario usando el browser siguiendo el patrón:

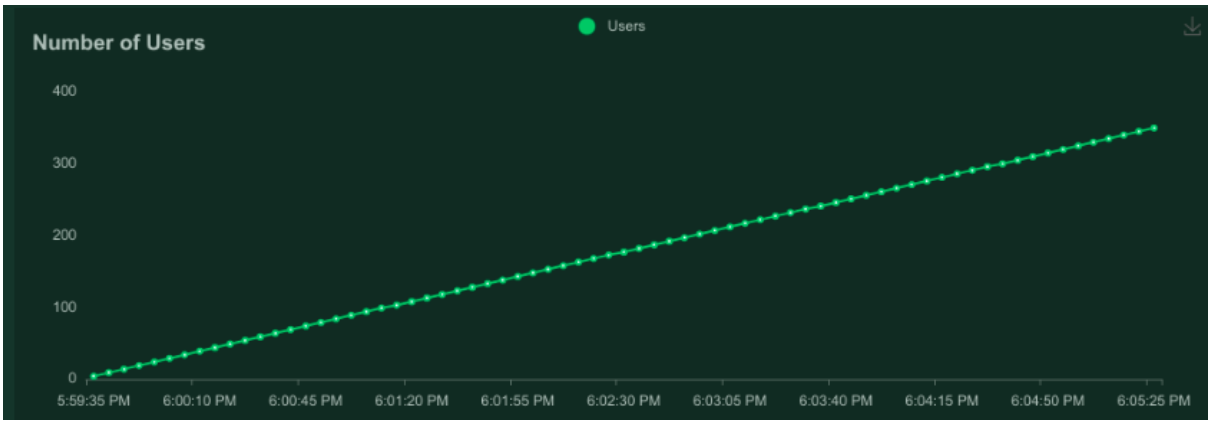
```
GET <página>.html
parseo de recursos (link, script, img)
descarga de los recursos (link, script, img)
POST /api/visits/<página>
GET  /api/visits/<página>
```

Los timeouts de las peticiones http realizadas, se basan en lo que establecemos como SLAs de los distintos recursos del sitio:

recurso	timeout	attempts(retry)
recursos html,img,script,link	32s	1
api/visits/<página>	4s	3

Asumimos que los usuarios están dispuestos a esperar que el sitio se pinte completamente, pero que una vez pintado no están dispuestos a esperar tanto tiempo por el contador.

Como estímulo al sistema se utilizó un patrón de carga lineal, con incremento de usuarios de 1 por segundo.



2.2 Resultados obtenidos

2.2.1 Preliminares: 1 instancia

Fecha:
8 de octubre del 2021 de 10:42 a 10:48

Configuración:
1 instancia para cada cloud function(**api, site, inc_visits**).

Resultados:

El punto de quiebre se da por timeouts en los servicios de **api** a los 90 usuarios. Se observa que una cantidad importante de requests son abortados por no haber instancias disponibles y si bien el tiempo de procesamiento disminuye durante la ejecución se producen timeouts.

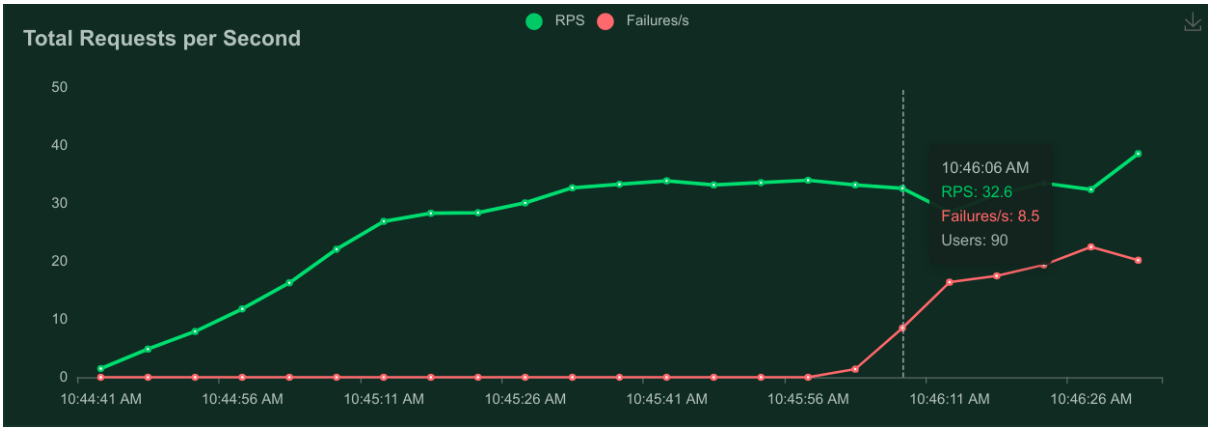
Por otro lado para el caso de los recursos html se observa que no hay tantas peticiones y que él mismo responde con http status code 304, es decir, se está aprovechando el caching de http para evitar el reprocesamiento, asumimos se está utilizando la caché interna de google.

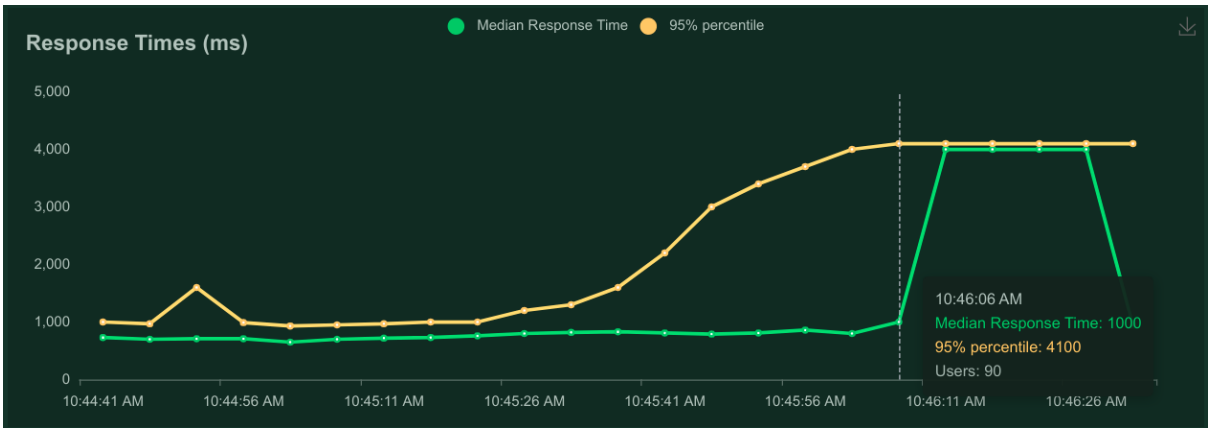
Por la corta duración de la prueba, no se puede sacar una conclusión fuerte sobre el origen del punto de quiebre, pero estamos orientados a decir que tiene que ver con el encolado de mensajes interno que hace GCP para atender los requests con Cloud Functions y que la aplicación no entro en calor.

De todas formas sirvió para descubrir que los recursos estáticos son cacheados por google.

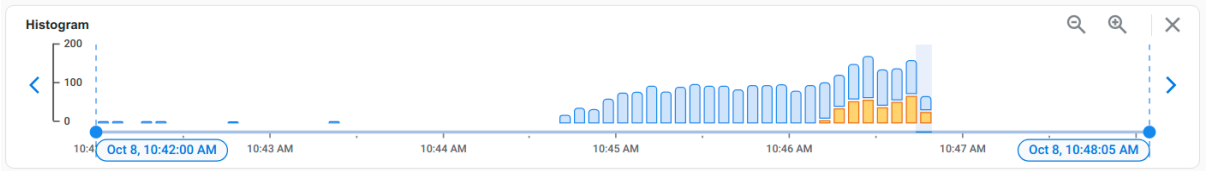
Datos:

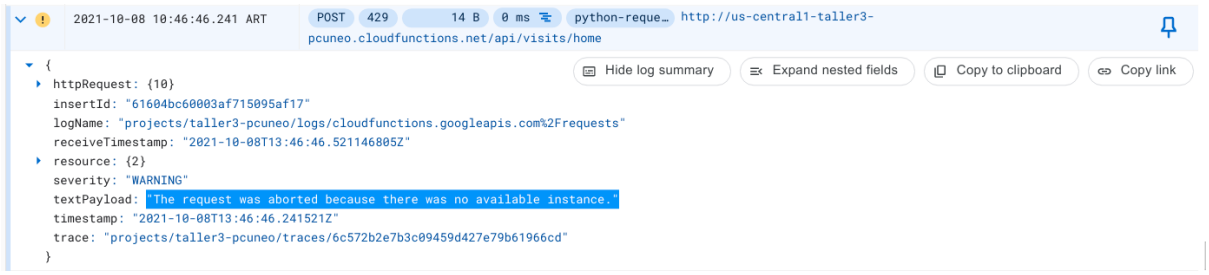
https://console.cloud.google.com/logs/query;timeRange=2021-10-08T13:44:00.000Z%2F2021-10-08T13:48:00.000Z;cursorTimestamp=2021-10-08T13:46:48.101211561Z?referrer=search&angularJsUrl=%2Flogs&project=taller3-pcuneo&query=%0A



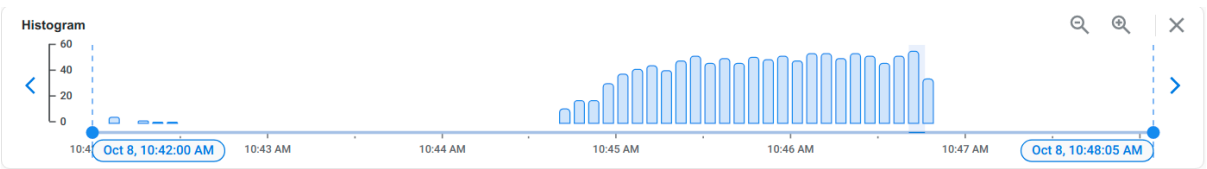


api:

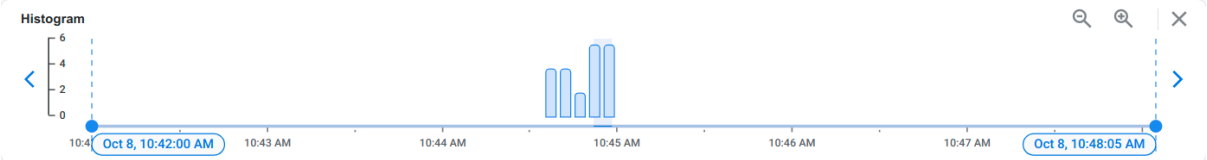


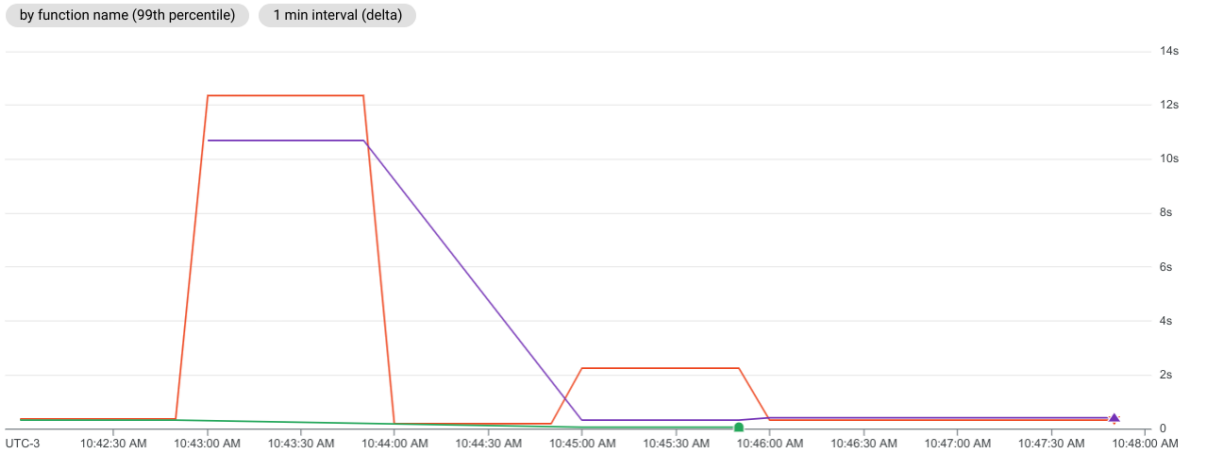


inc_visits:



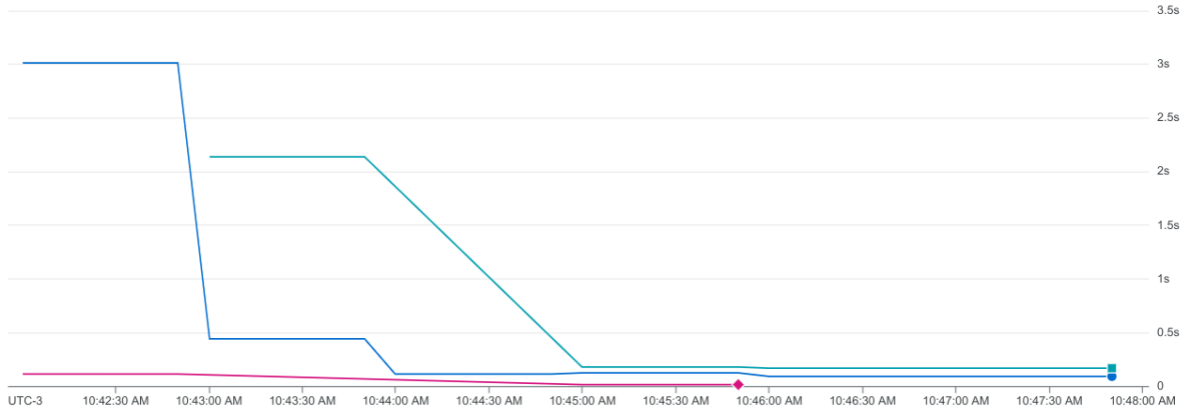
site:





function_name	Value
api	0.31s
inc_visits	0.39s
site	-

by function name (mean) | 1 min interval (delta)



function_name	Value
api	0.098s
inc_visits	0.169s
site	-

2.2.2 Prueba Control

Fecha:
10 de octubre del 2021 de 19:19 a 19:35

Configuración:
3 instancias cada función, sin shardeo, sin timeout en locust.

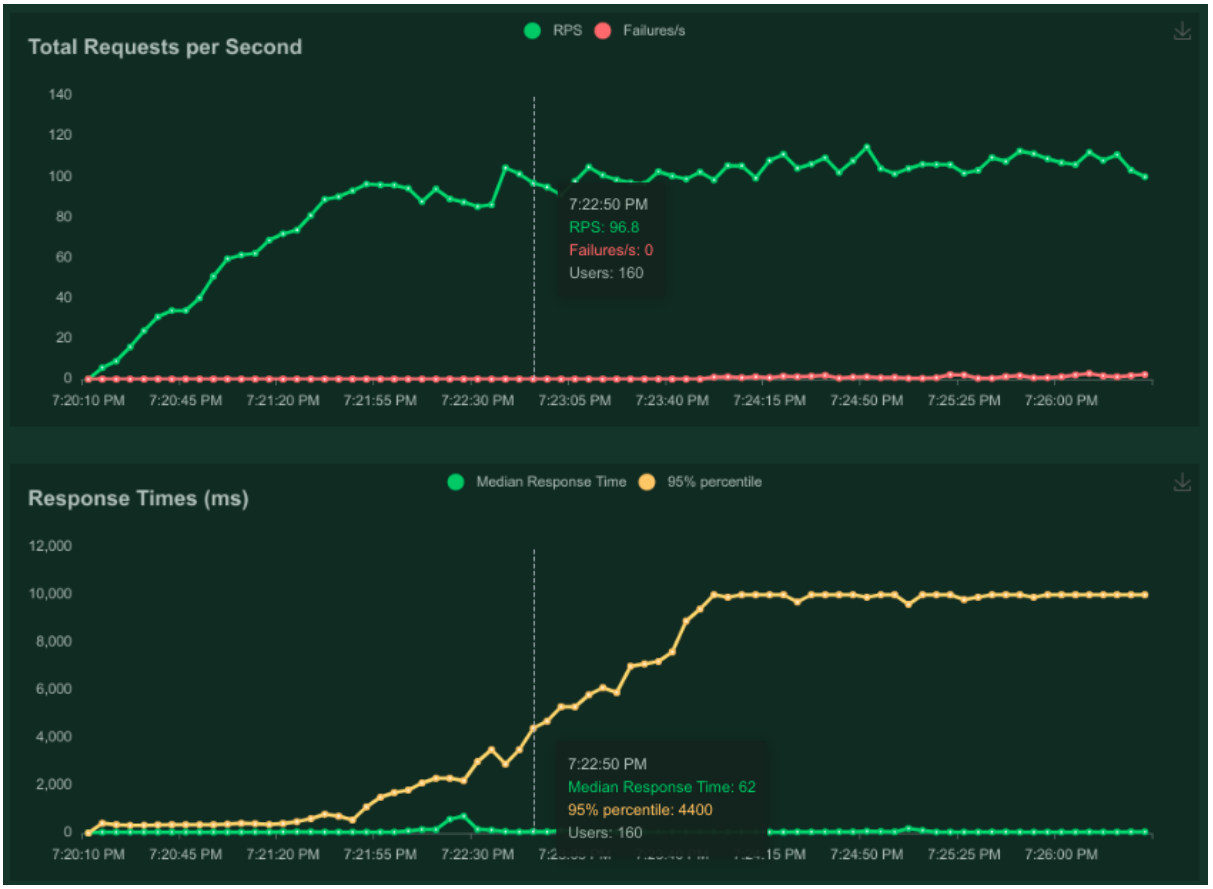
Resultados:
Al incrementar la cantidad de instancia de api, se pueden atender un número mayor de request, pero también se aumenta la cantidad de eventos de pub/sub a procesar por inc_visits.

Vemos que con 3 instancias el punto de quiebre se produce a los 160 usuarios, mientras que cada request se atiende en menos de 1 segundos, creemos que el ruteo interno de google sumado al roundtrip hace demorar el tiempo de respuesta por arriba de los 4s.

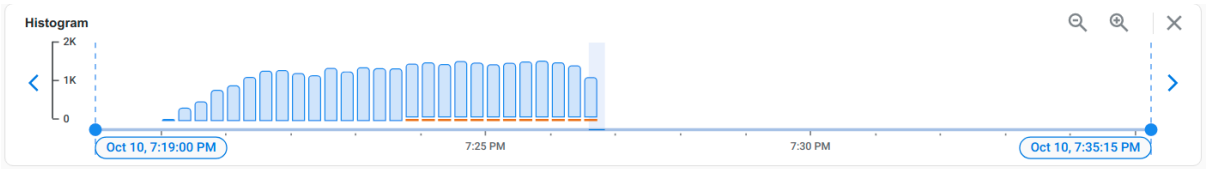
Por otro lado en este caso que se acumulan mas eventos en el pub/sub se empiezan a producir errores primero por que al estar limitada a 3 instancias, no se pueden atender todos los eventos con celeridad, por otro lado al haber 3 instancias empezamos a ver errores de concurrencia sobre los contadores, es decir, hay al menos 2 transacciones que tratan de escribir sobre el mismo contador y uno termina siendo abortada.
Afortunadamente ambos casos de error son re-intentados por la función que procesa los eventos, por lo cual no hay fallos en la correctitud, sin embargo muchos eventos tienen que ser procesados 2 veces por lo que aumenta el tiempo total de procesamiento.

Para el servicio de archivos estáticos, vemos que el cache interno de google funciona de forma óptima y evita procesar múltiples veces un archivo que no tuvo cambios.

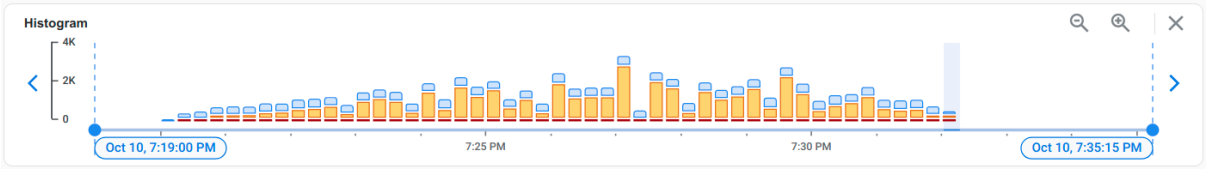
Datos:
https://console.cloud.google.com/logs/query;query=;timeRange=2021-10-10T22:19:00.000Z%2F2021-10-10T22:35:00.000Z;cursorTimestamp=2021-10-10T22:32:13.142432518Z?referrer=search&angularJsUrl=%2Flogs&project=taller3-pcuneo&query=%0A



api: 236 warnings, GET 5528 POST 5659

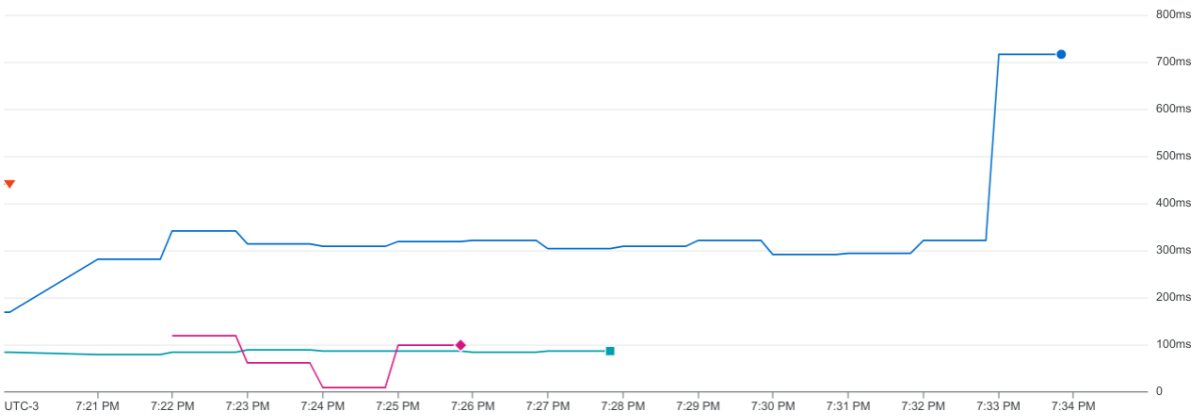


inc_visits: 237 errores



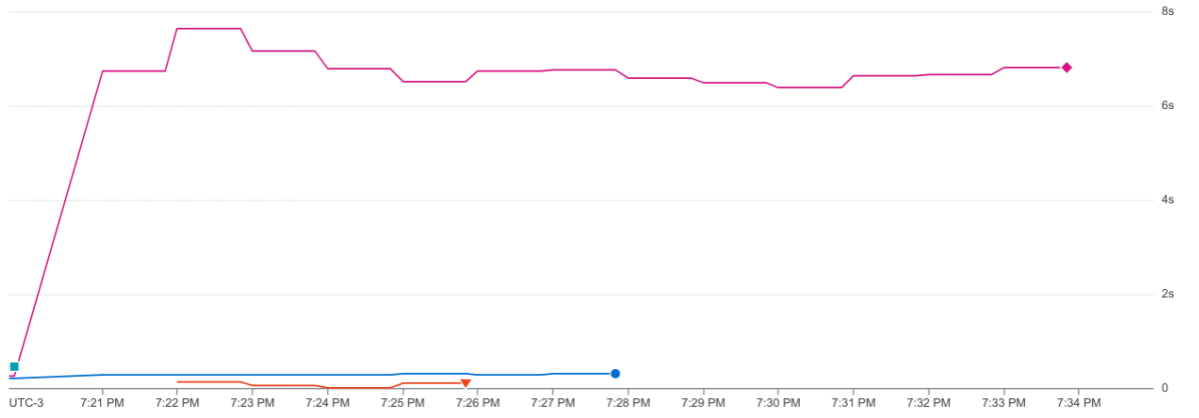
site:

by function name (mean) | 1 min interval (delta)



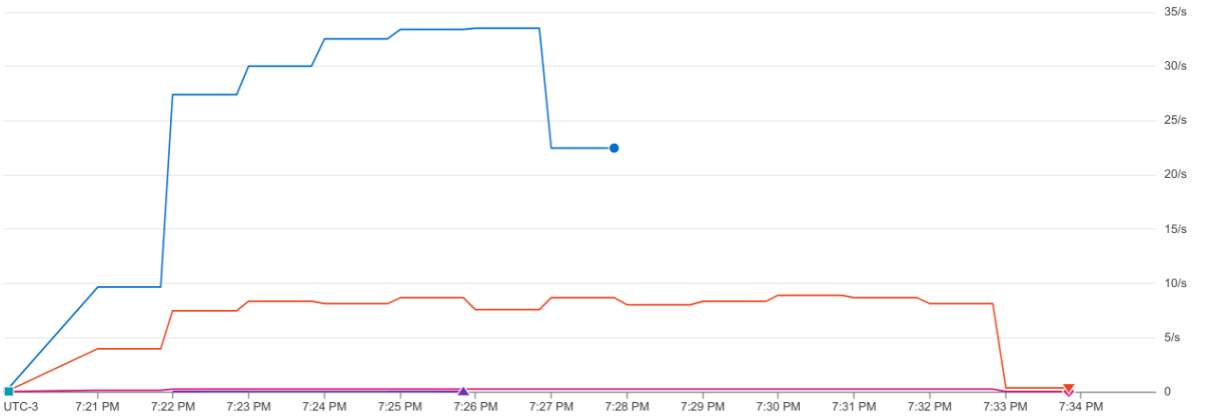
function_name	Value
api	-
config	-
inc_visits	715.75ms
site	-

by function name (99th percentile) | 1 min interval (delta)



function_name	Value
api	-
config	-
inc_visits	6.48s
site	-

1 min interval (rate)



function_name	status	trigger_type	Value
api	ok	HTTP_TRIGGER	0.42/s
config	ok	HTTP_TRIGGER	0.02/s
inc_visits	crash	google.pubsub.topic.publish	0.02/s
inc_visits	ok	google.pubsub.topic.publish	0.2/s

2.3 Optimizaciones de Performance

2.3.1 Prueba Sharding Counter

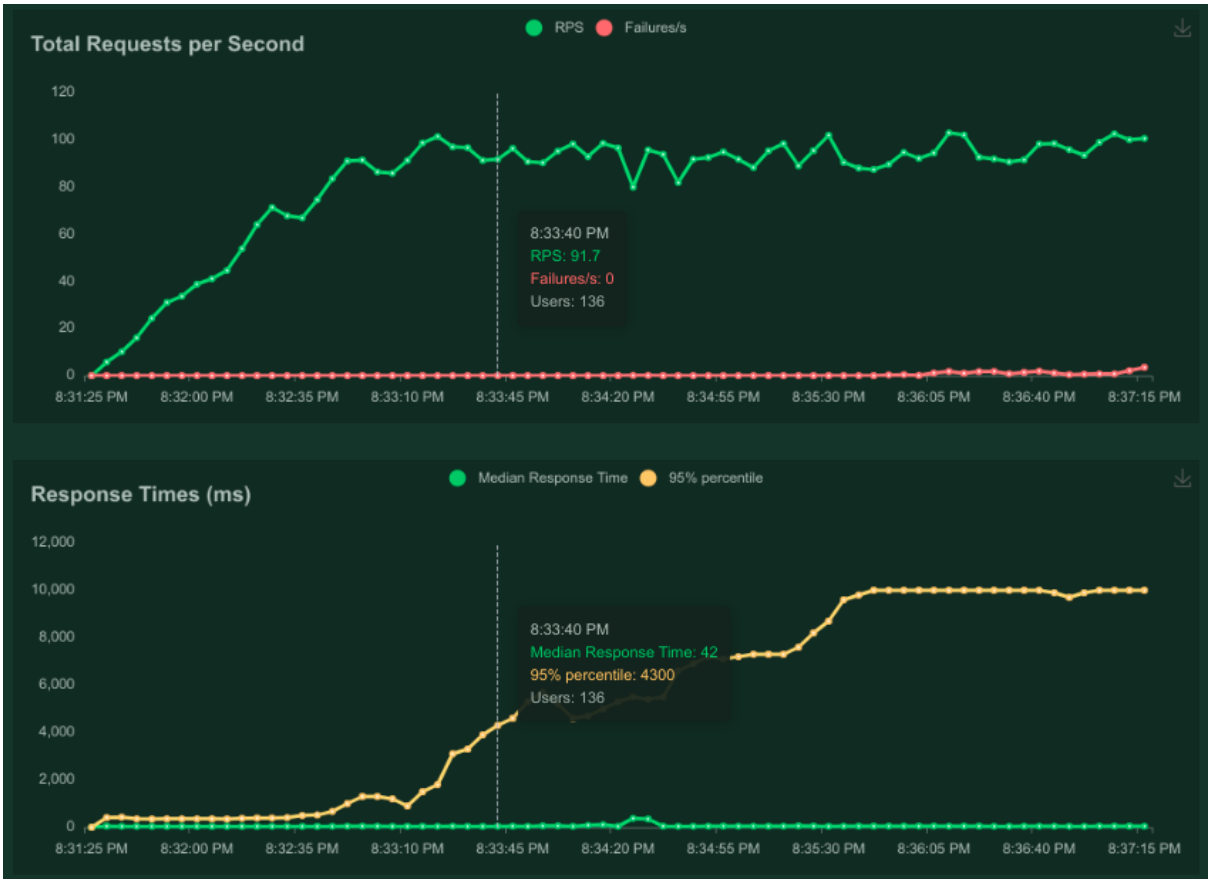
Fecha:
10 de octubre del 2021 de 20:31 a 20:40

Configuración:
3 instancias cada función, usando sharding de 10 partes, y bulk get de las entidades.

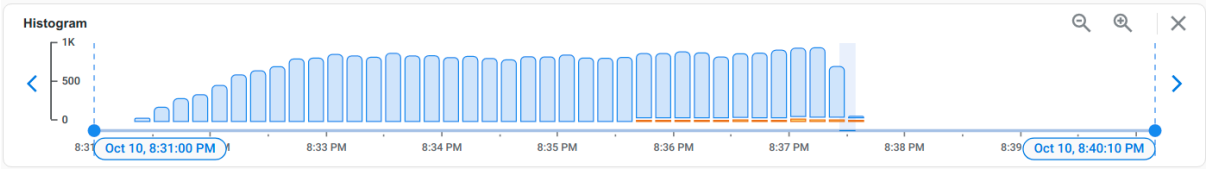
Resultados:
Al utilizar contadores shardeados, podemos ver una mejoría significativa en el tiempo total para procesar los eventos de pub/sub, y una disminución de la cantidad de errores.
Esto se debe a que al escribir en distintas entidades(aunque sea de forma aleatoria), se disminuyen la cantidad de transacciones en conflicto, y por lo tanto hay menos eventos que re-intentar. El tiempo de procesamiento interno de las funciones no cambia mucho comparado con el contador no shardeado, pero al haber menos errores procesar todos los eventos toma menos tiempo.

Respecto del procesamiento de requests en la función de api, no se observa mucha diferencia, teniendo en cuenta que se está utilizando una query para recuperar todas los shards que corresponden a un contadores, es decir, se hace bulk get.

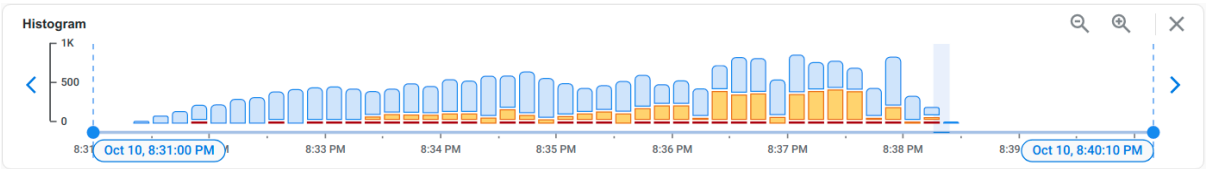
Datos:
https://console.cloud.google.com/logs/query;query=;timeRange=2021-10-10T23:31:00.000Z%2F2021-10-10T23:40:00.000Z;cursorTimestamp=2021-10-10T23:38:20.213137387Z?referrer=search&angularJsUrl=%2Flogs&project=taller3-pcuneo&query=%0A



api: 160 warnings

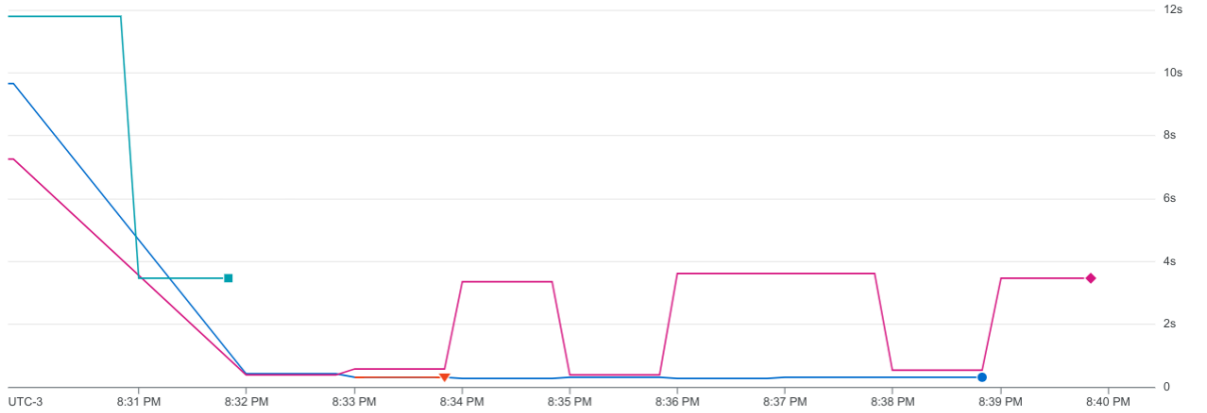


inc_visits: 66 errores



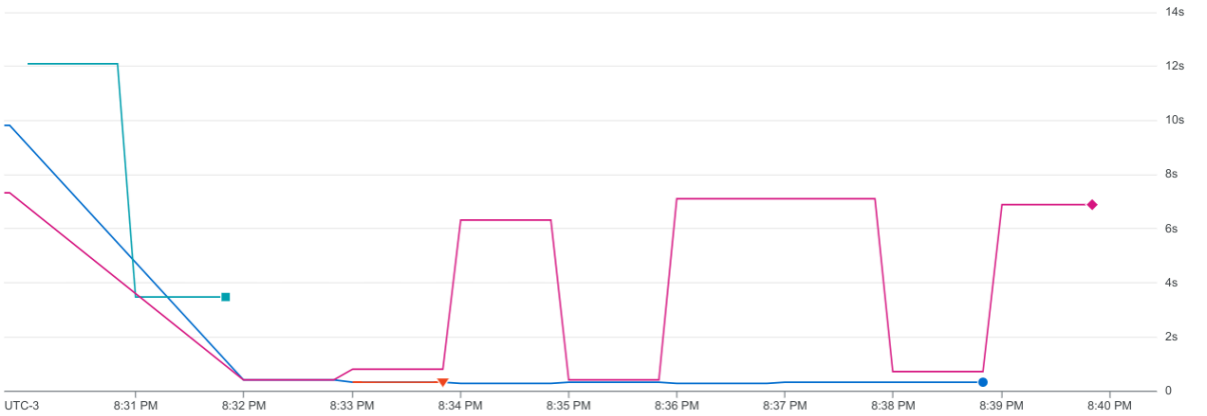
site:

by function name (mean) | 1 min interval (99th percentile)



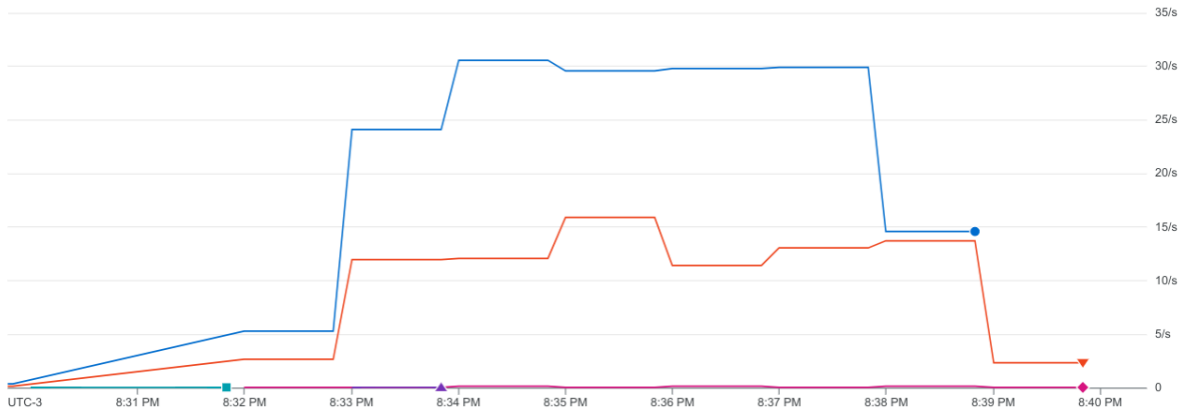
function_name	Value
api	-
config	-
inc_visits	3.48s
site	-

by function name (99th percentile) | 1 min interval (99th percentile)



function_name	Value
api	-
config	-
inc_visits	6.89s
site	-

1 min interval (rate)



function_name	status	trigger_type	Value
api	ok	HTTP_TRIGGER	-
config	ok	HTTP_TRIGGER	-
inc_visits	crash	google.pubsub.topic.publish	0.02/s
inc_visits	ok	google.pubsub.topic.publish	2.35/s

2.3.2 Prueba Sharding + Instancias de api

Fecha:
11 de octubre de 2021 de 10:59 a 11:04
Configuración:
16 instancias de api, con contador shardeado y bulk get de los shards.

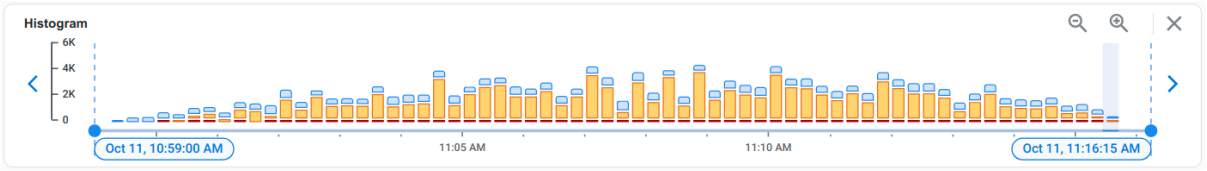
Resultados:
Al incrementar la cantidad de instancias de api a 16 vemos que el sistema llega a un punto de quiebre visible por el usuario. Es decir, todos los request a api responden dentro del tiempo aceptable.
Esto quiere decir que los timeouts que vimos en corridas anteriores están fuertemente influenciados por la forma en que GCP maneja el escalado automático y cómo bufferea los request, podemos decir esto por que en la corridas anteriores el tiempo de procesamiento era menor a 4 segundos, pero aun así se producían timeouts.

Como ya se observó en corridas anteriores al encolar más eventos de pub/sub a mayor velocidad, aumentan la cantidad de errores debido a la restricción del recurso y a que aumenta las probabilidades de colisión de las transacciones.
Y también aumenta el tiempo total para procesar los eventos debido a las colisiones y errores.

Datos:
https://console.cloud.google.com/logs/query;query=;timeRange=2021-10-11T13:59:00.000Z%2F2021-10-11T14:16:00.000Z;cursorTimestamp=2021-10-11T14:15:38.555173267Z?referrer=search&angularJsUrl=%2Flogs&project=taller3-pcuneo&query=%0A

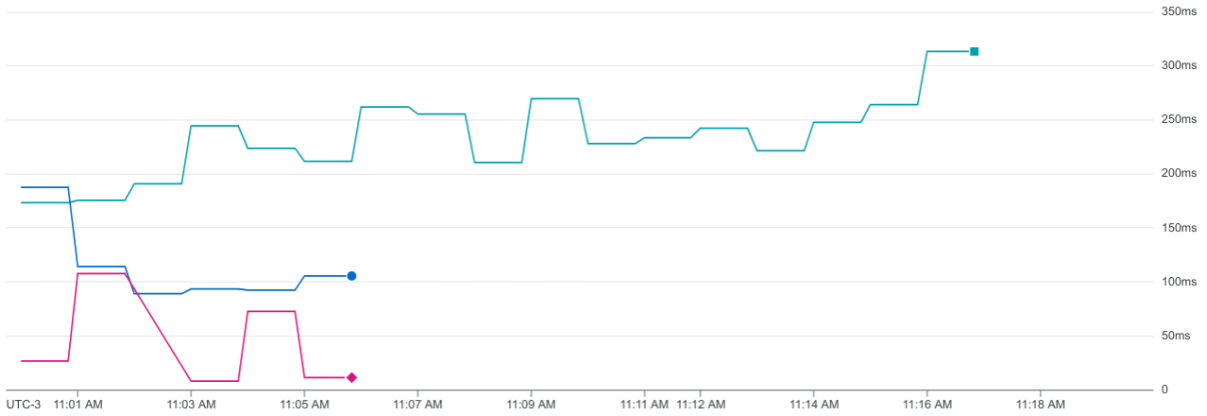
api

inc_visits:156 errores



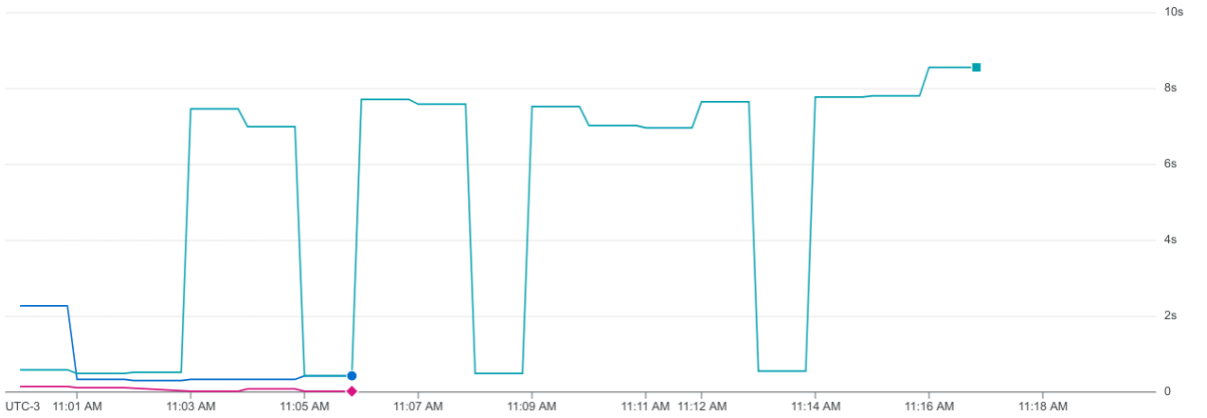
site:

by function name (mean) | 1 min interval (delta)



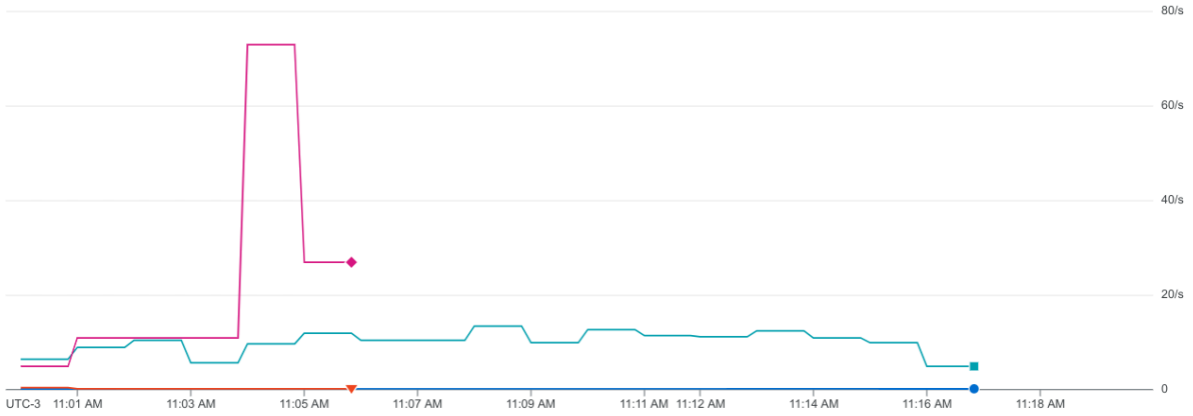
function_name	Value
api	-
inc_visits	313.18ms
site	-

by function name (99th percentile) | 1 min interval (delta)



function_name	Value
api	-
inc_visits	8.54s
site	-

1 min interval (rate)



function_name	status	trigger_type	Value
api	ok	HTTP_TRIGGER	-
inc_visits	crash	google.pubsub.topic.publish	0.07/s
inc_visits	ok	google.pubsub.topic.publish	4.87/s
site	ok	HTTP_TRIGGER	-

2.3.3 Prueba Batch Increment + Heap Cache

Fecha:
11 de octubre de 2021 de 17:59 a 18:06

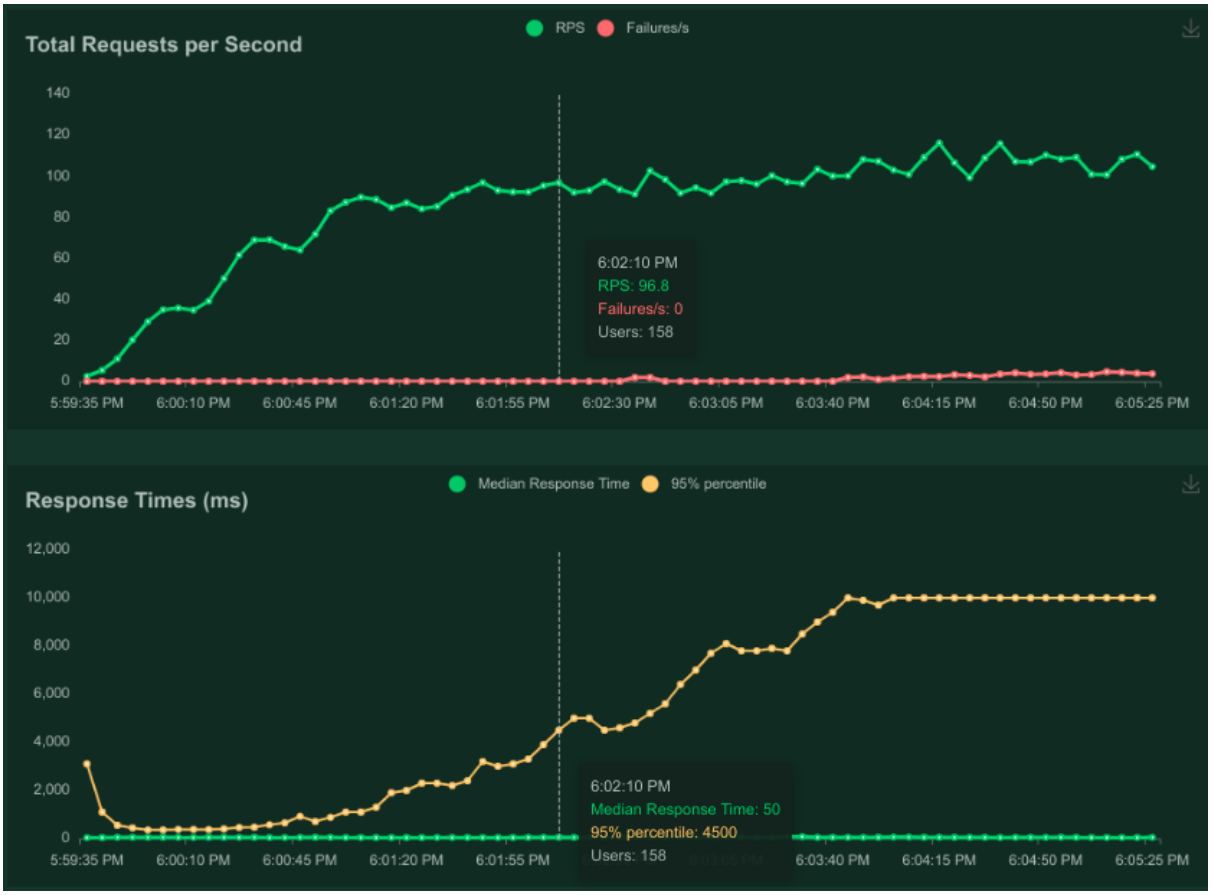
Configuración:
3 instancias de cada función, inc_visits haciendo batching(utilizando un mapa en memoria para acumular los cambios antes de ponerlos a datastore) y sin shardear los contadores.

Tech Debt Note:
Vale aclarar que esta implementación no es tolerante a fallos ya que en casos de crashear la función se pierden datos.

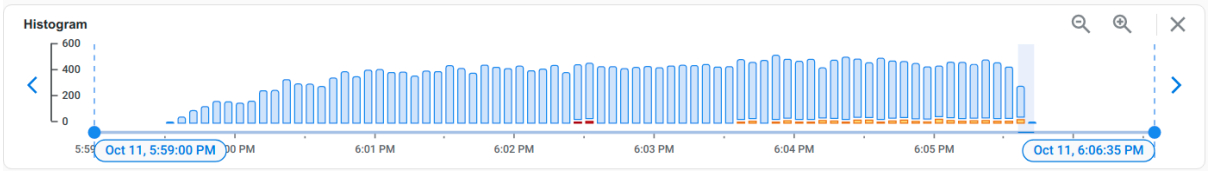
Resultados:
El comportamiento de api es similar al caso de control el tiempo de procesamiento es menor al segundo pero debido a la restricción de recursos pasados los 150 usuarios se empiezan a producir timeouts.

Como es de esperar el procesamiento de eventos de pub/sub mejora significativamente, no se registran errores, y el tiempo total de procesamiento disminuye a menos de 1 segundo.

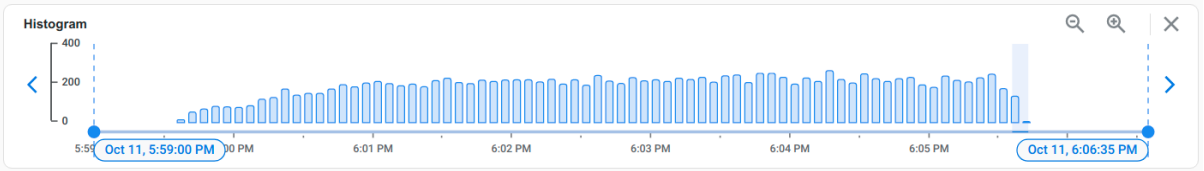
Datos:
https://console.cloud.google.com/logs/query;query=;timeRange=2021-10-11T20:59:00.000Z%2F2021-10-11T21:06:30.000Z;cursorTimestamp=2021-10-11T21:05:40.003649508Z?referrer=search&angularJsUrl=%2Flogs&project=taller3-pcuneo&query=%0A



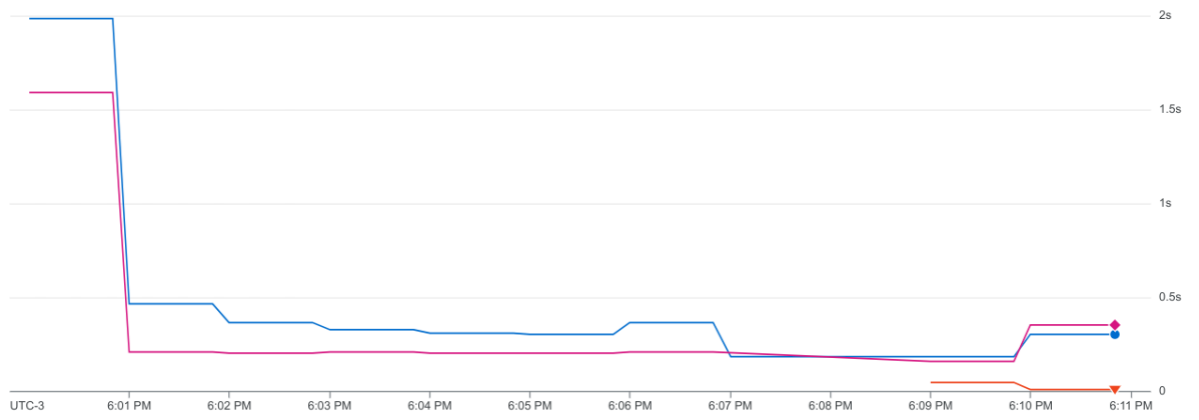
api: 19 errores 402 warnings



inc_visits: 0 warnings

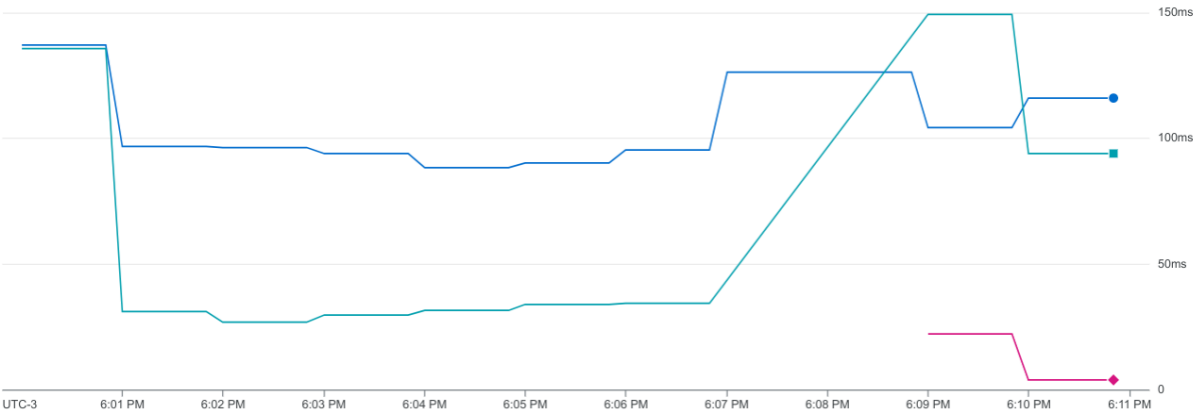


by function name (99th percentile) | 1 min interval (delta)



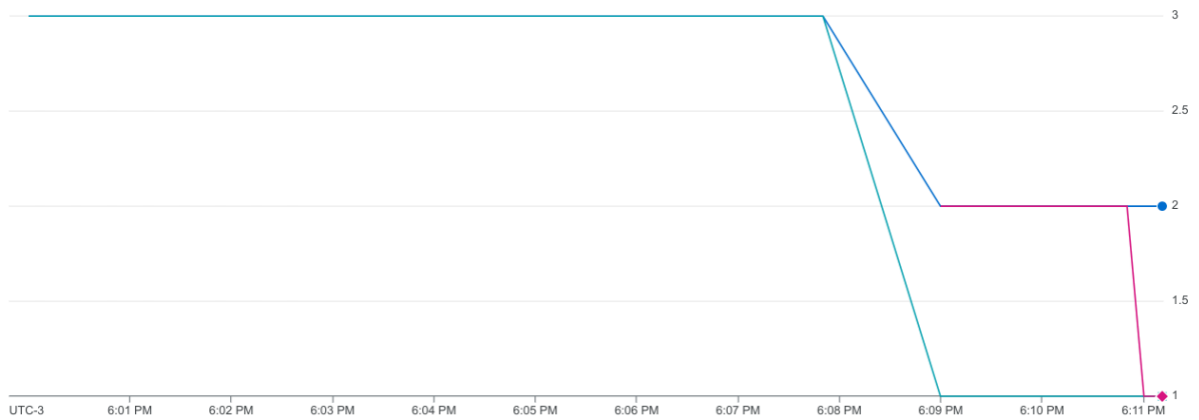
function_name	Value
api	0.363s
inc_visits	0.2s
site	-

by function name (mean) | 1 min interval (delta)



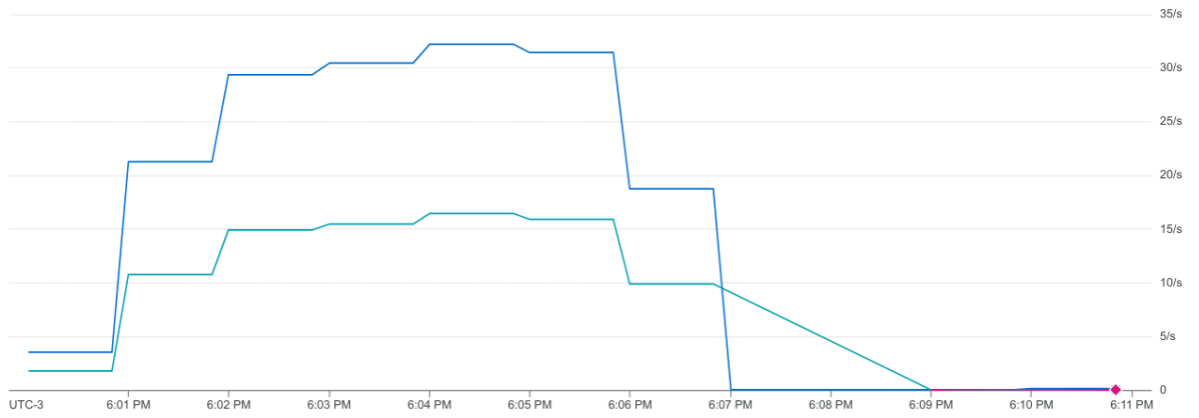
function_name	Value
api	116.14ms
inc_visits	93.87ms
site	3.92ms

by function name (max) | 1 min interval (mean)



function_name	Value
api	3
inc_visits	3
site	-

1 min interval (rate)



function_name	trigger_type	Value
api	HTTP_TRIGGER	0.17/s
inc_visits	google.pubsub.topic.publish	0.08/s
site	HTTP_TRIGGER	0.02/s

3 Conclusión

Desde la perspectiva de los tiempos de respuesta, vemos que el sistema escala horizontalmente agregando más funciones.

Desde la perspectiva de la consistencia de los contadores, vemos que al shardear mejora el tiempo de convergencia y mucho mejor aún si se hace batching de los updates.

El sharding del contador de todas formas sigue teniendo transacciones que colisionan, y por tanto se deben procesar 2 veces.

El batching de contador mejora mucho la performance, pero la implementación presentada no es tolerante a fallos. La cuestión con implementarlo de forma tolerante es que implica guardar el acumulador en storage estable por cada ítem procesado con lo cual se perdería la optimización. En este caso se puede realizar un compromiso entre la performance y el tamaño de batch dispuesto a perder en una eventual falla.

Se omite la prueba con recursos ilimitados ya que a priori no parece un caso real para un contador. Un contador de visitas en general no sería pieza crítica de un negocio por estos días, sin embargo de tener que llevar la misma arquitectura a otro negocio donde el procesamiento sea crítico habría que realizar el cálculo costo beneficio del escalado infinito contra el procesamiento con timeouts más altos y tiempo total de convergencia también más alto.