Arquitectura de Software

Grupo: **Bob el Constructor**

TP1

75.73 - Arquitectura de Software

FIUBA - 2C2020

Integrantes

Nombre y apellido	**Padrón**
Javier Ferreyra	**100680**
Bautista Canavese	**99714**
Sebastian Loguercio	**100517**
Santiago Mariani	**100516**

Sección 1 **2**

Ping 2

Gunicorn 2

10 usuarios por segundo 2

100 usuarios por segundo 3

400 usuarios por segundo 5

Endurance 6

Node 8

10 usuarios por segundo 8

100 usuarios por segundo 9

400 usuarios por segundo 9

Endurance 10

Node Replicado 11

400 usuarios por segundo 11

Endurance 11

Impacto en atributos de calidad 12

Proxy 13

Gunicorn 13

Un request cada 5 segundos 13

Un request cada 3 segundos 14

Node 16

Un request cada 3 segundos 16

Rampa 17
Node Replicado 19
Rampa 19
Intensive 22
Gunicorn 22
Un request cada 5 segundos 22
Un request cada 3 segundos 23
Node 25
Un request cada 5 segundos 25
Un request cada 3 segundos 26
Node Replicado 28
Un request cada 3 segundos 28
Impactos en atributos de calidad (Intensive) 29
Sección 2 **31**
Tiempo de respuesta 31
Servidor Sincrónico o Asincrónico 32
Cantidad de workers 34
Resumen 36
Link al repositorio **36**

Sección 1

Ping

A continuación se describirá la ejecución de distintos escenarios de prueba realizados para ver cómo se comporta el endpoint Ping: representa un valor constante y rápido, bajo distintos deploys. Comenzando por el webserver hecho con Gunicorn, con un único worker, se muestra la ejecución de distintos escenarios.

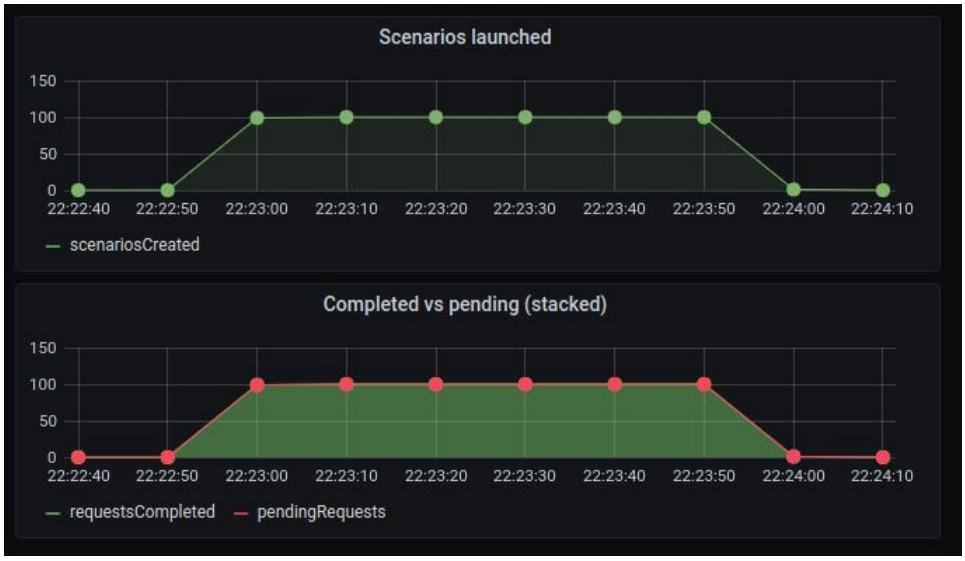
Se realizaron cuatro pruebas, incrementando la cantidad de usuarios por segundo, a continuación se muestran los resultados.

Gunicorn

10 usuarios por segundo

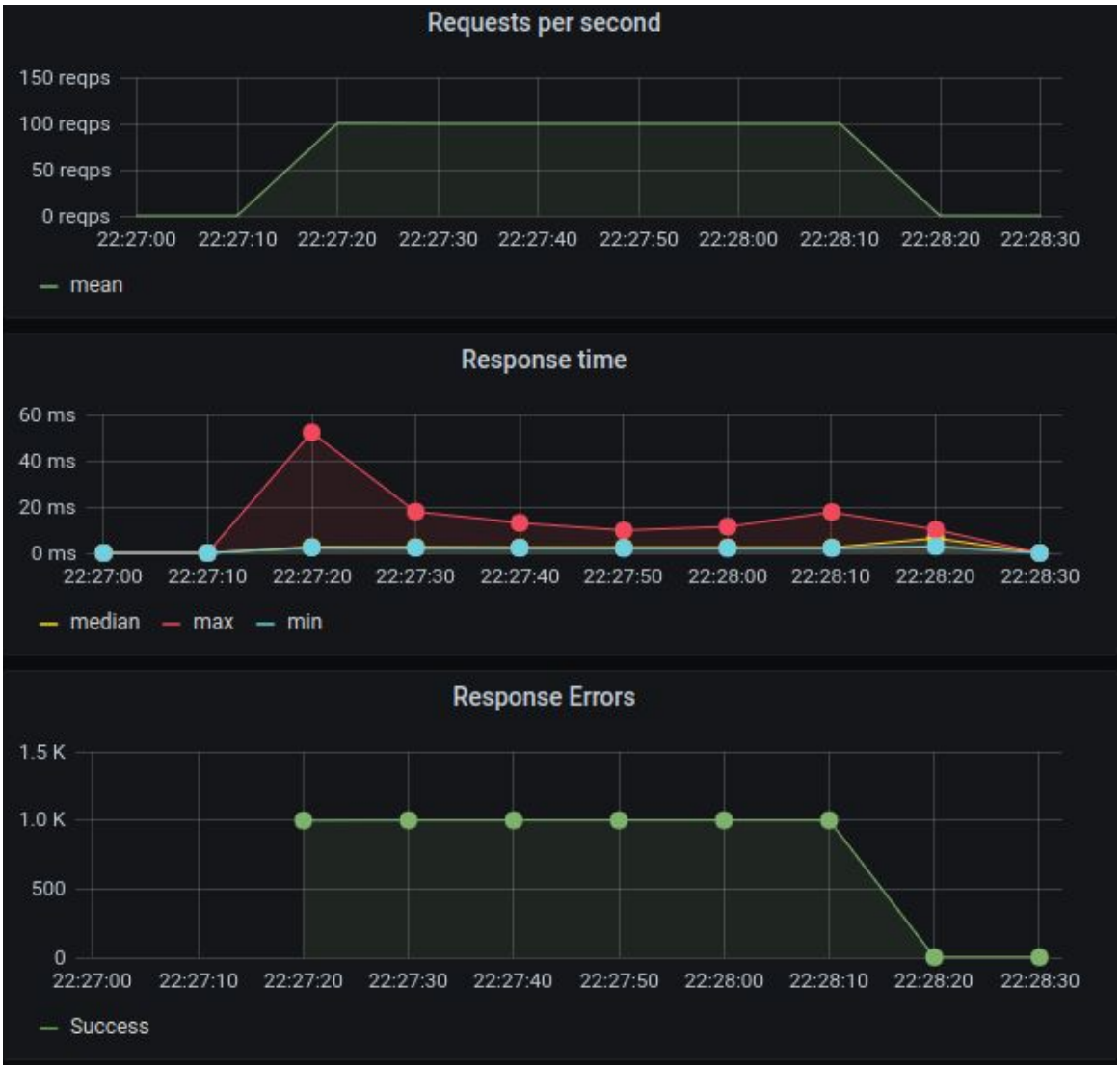
El primer test consiste en 10 usuarios por segundo durante 1 minuto, vemos que el servidor lo puede manejar sin problemas. En los tiempos de respuesta hay algunos máximos que se separan de la media pero siguen siendo tiempos bastante aceptables que no superan los 40 ms.

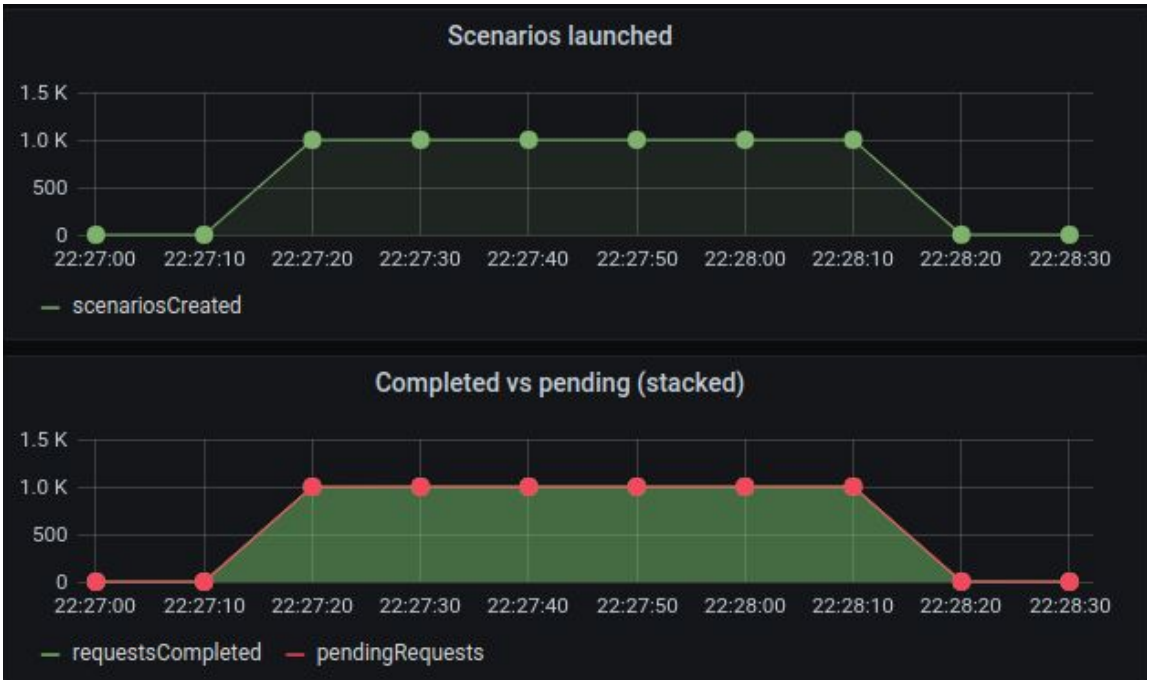

Metrics
Requests per second
15 reqps
10 reqps
5 reqps
0 reqps
22:22:40 22:22:50 22:23:00 22:23:10 22:23:20 22:23:30 22:23:40 22:23:50 22:24:00 22:24:10
mean
Response time
40 ms
30 ms
20 ms
10 ms
0 ms
22:22:40 22:22:50 22:23:00 22:23:10 22:23:20 22:23:30 22:23:40 22:23:50 22:24:00 22:24:10
median max min
Response Errors
150
100
50
0
22:22:40 22:22:50 22:23:00 22:23:10 22:23:20 22:23:30 22:23:40 22:23:50 22:24:00 22:24:10
Success


Scenarios launched
150
100
50
0
22:22:40 22:22:50 22:23:00 22:23:10 22:23:20 22:23:30 22:23:40 22:23:50 22:24:00 22:24:10
scenariosCreated
Completed vs pending (stacked)
150
100
50
0
22:22:40 22:22:50 22:23:00 22:23:10 22:23:20 22:23:30 22:23:40 22:23:50 22:24:00 22:24:10
requestsCompleted pendingRequests

100 usuarios por segundo

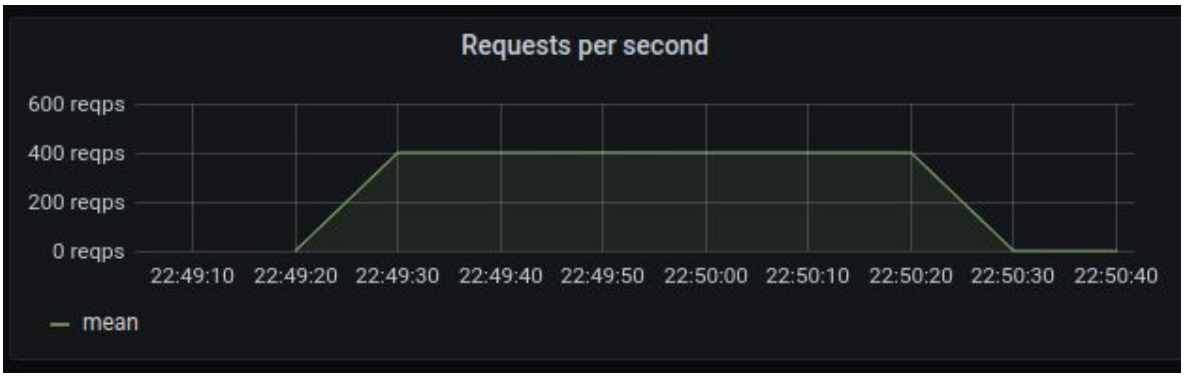
En este caso también se ve que no se detectan problemas con el servidor y puede responder perfectamente.

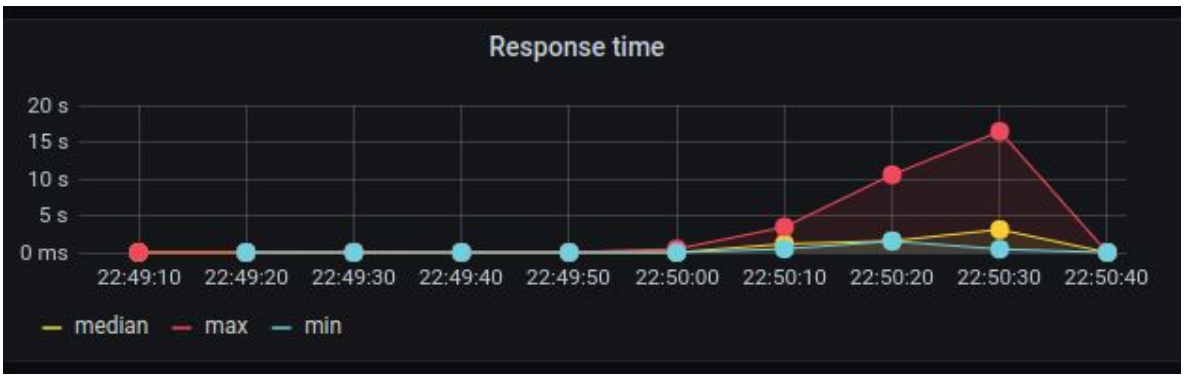


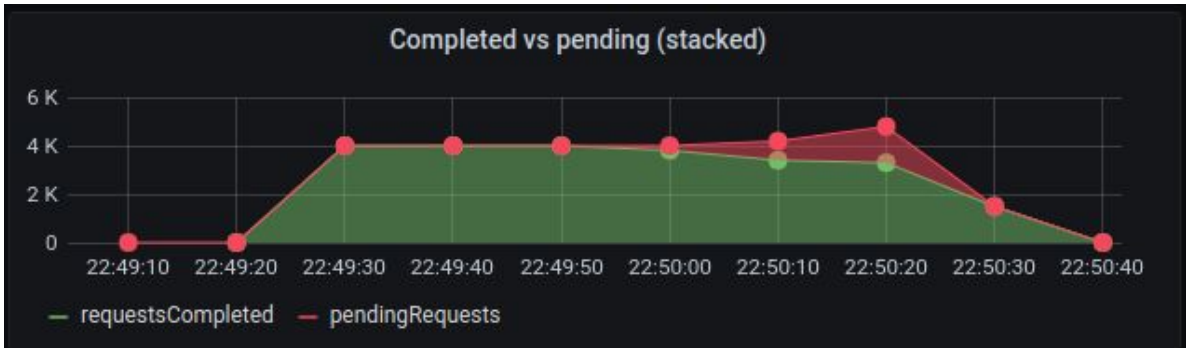


400 usuarios por segundo

En este caso sí notamos una degradación en la performance al mantener por un tiempo relativamente prolongado una mayor cantidad de arribos de usuarios por segundo.





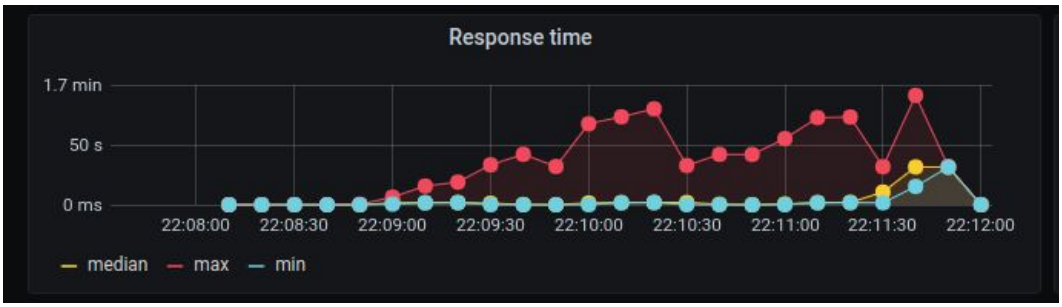


En los gráficos podemos observar que por el final comenzamos a tener requests que tardan algunos segundos en completarse, con máximos en casi 20 segundos ya que el servidor no puede mantener el ritmo y se comienzan a acumular las requests en el worker.

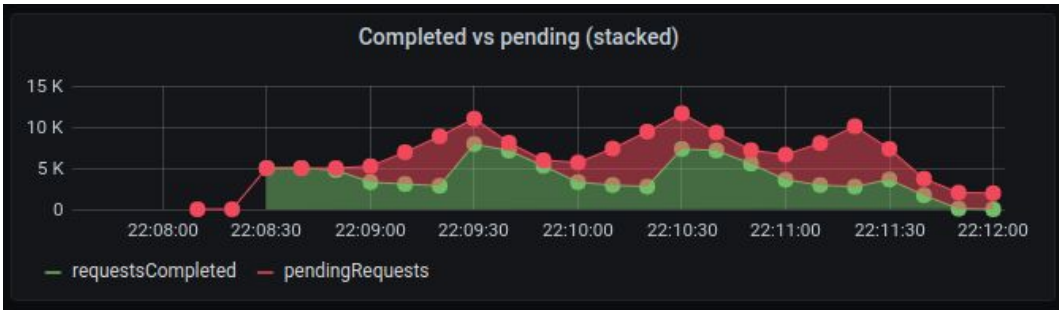
En promedio vemos que el tiempo de respuesta se mantiene, pero tendremos ciertos usuarios que pueden percibir una mala performance ya que sus requests tendrán un tiempo de respuesta bastante alto.

Endurance

En este caso tenemos un escenario similar al anterior pero con 500 usuarios por segundo por un tiempo más prolongado.



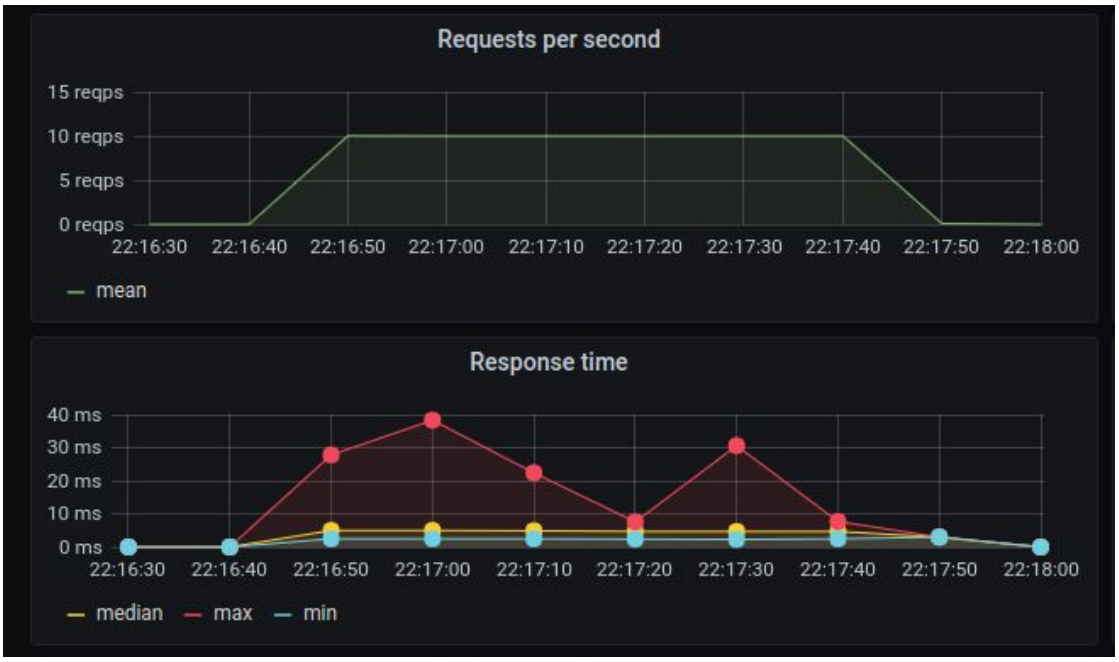
Si nos concentramos en el response time vemos que la performance comienza a degradarse también en 1 minuto como ocurría en el caso anterior, con máximos mayores a 1 minuto, por lo que aquí también ocurre que algunos usuarios comienzan a percibir una degradación en la performance. La diferencia es que luego de varios minutos la mayoría de los usuarios comienzan a percibir esta degradación.


Completed vs pending (stacked)
15 K
10 K
5 K
0
22:08:00
22:08:30
22:09:00
22:09:30
22:10:00
22:10:30
22:11:00
22:11:30
22:12:00
requestsCompleted
pendingRequests

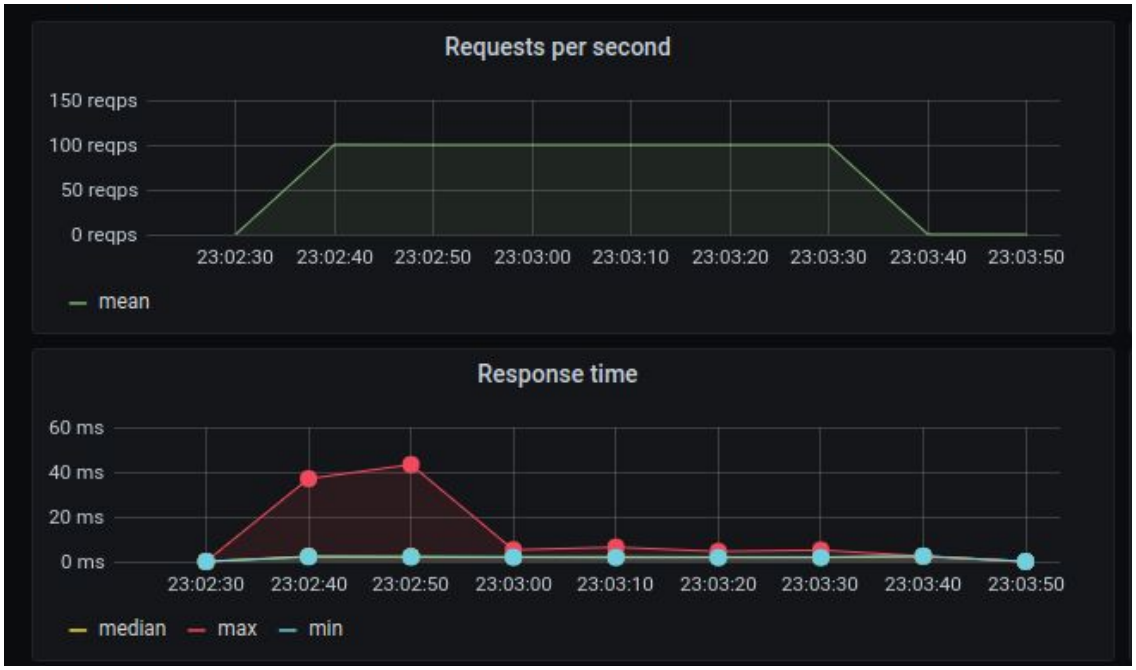
Node

Para el servidor implementado con Node.js se realizaron las mismas pruebas, obteniendo en general resultados similares a los obtenidos anteriormente con Gunicorn, ya que estamos trabajando sobre un endpoint bastante simple. A continuación vamos a presentar los gráficos donde se puede observar que es similar a los mostrados antes con Gunicorn.

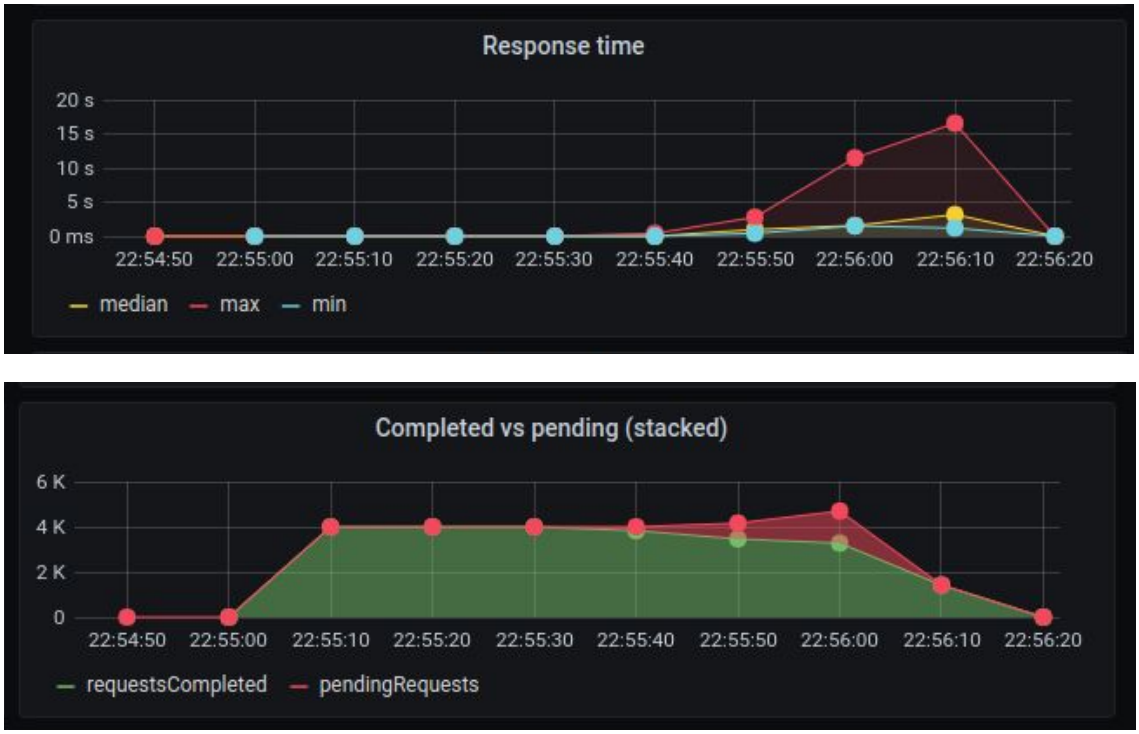
10 usuarios por segundo



100 usuarios por segundo

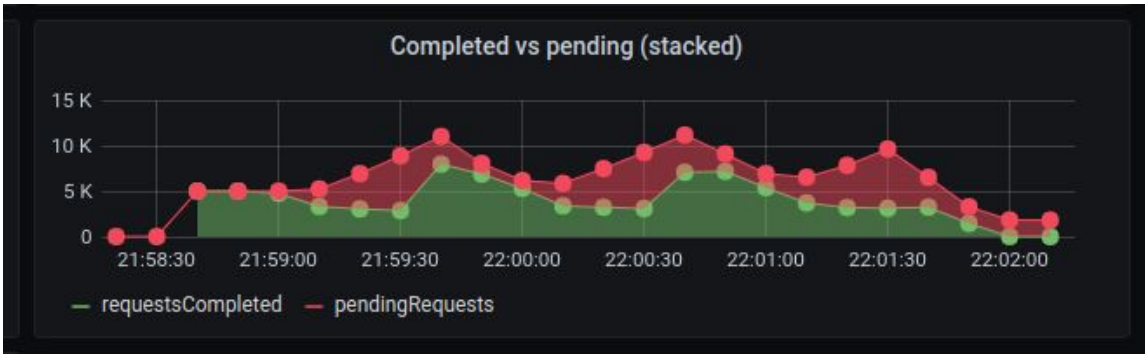


400 usuarios por segundo



Endurance


Response time
1.7 min
50 s
0 ms
21:58:30
21:59:00
21:59:30
22:00:00
22:00:30
22:01:00
22:01:30
22:02:00
median
max
min

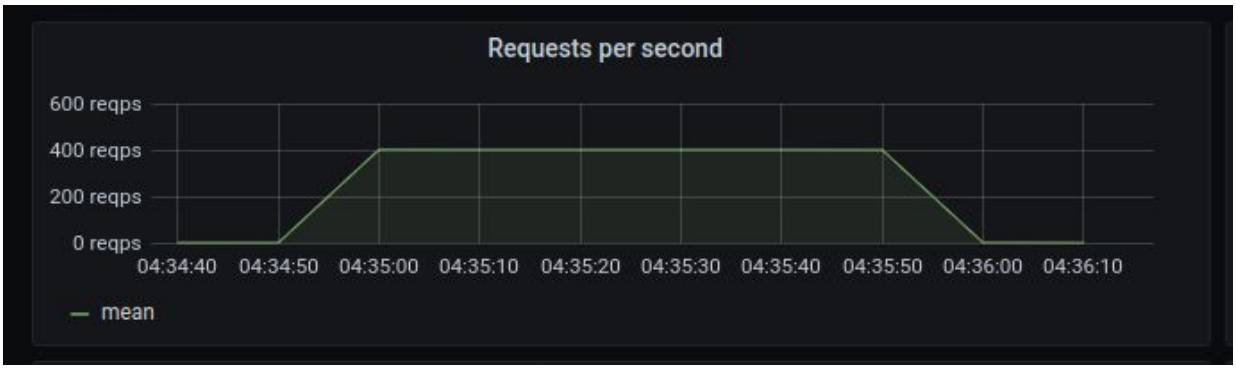

Completed vs pending (stacked)
15 K
10 K
5 K
0
21:58:30
21:59:00
21:59:30
22:00:00
22:00:30
22:01:00
22:01:30
22:02:00
requestsCompleted
pendingRequests

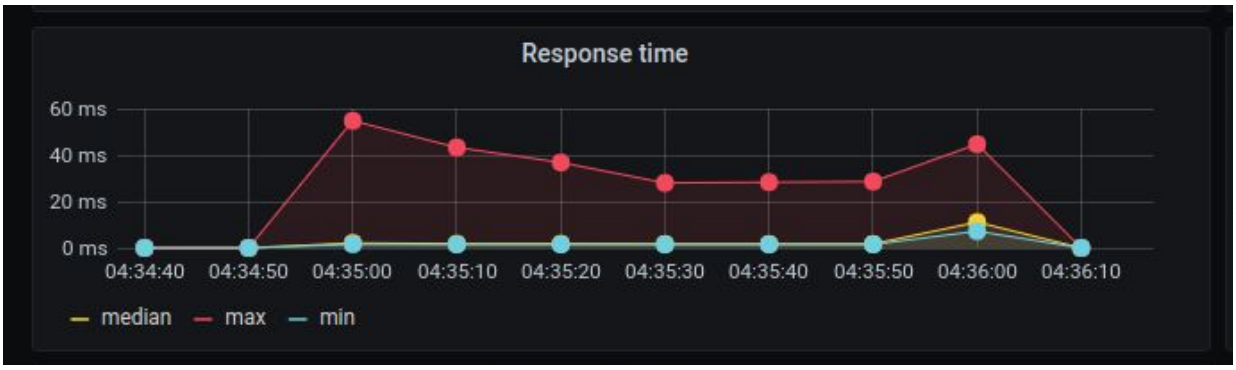
Node Replicado

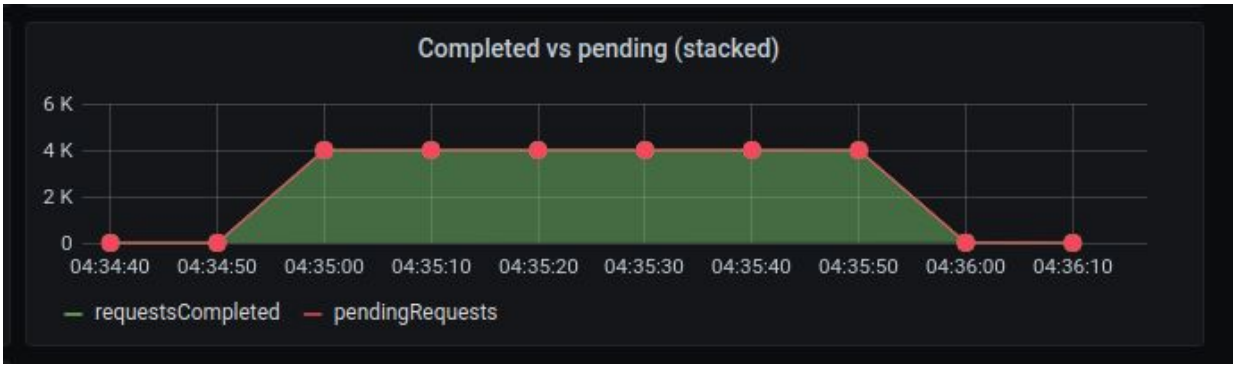
Si replicamos el contenedor de Node para correr 3 instancias, podemos ver que se obtiene una importante mejora en la performance.

400 usuarios por segundo

En este caso, vemos que los resultados de 400 usuarios por segundo son comparables a los obtenidos con 100 usuarios en el caso anterior, con tiempos de respuesta que no superan los 60 ms.

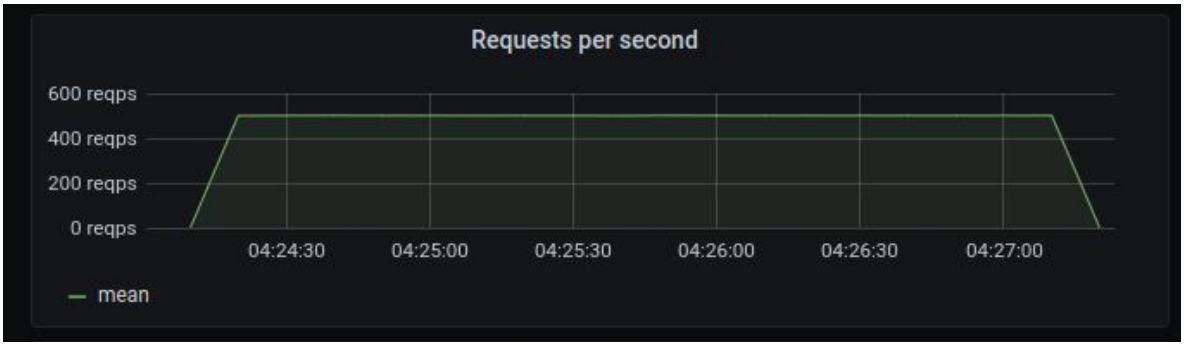






Endurance

En este caso también podemos observar una importante mejora, ya que al mantener una cantidad alta de usuarios por segundo durante un tiempo prolongado no vemos que haya una degradación en la performance a medida que pasa el tiempo, nuevamente manteniendo los tiempos de respuesta por debajo de los 60 ms.



Impacto en atributos de calidad

Viendo los gráficos de tiempo de respuesta, en general al aumentar las requests por segundo llega un momento que el servidor se satura, alcanzando lógicamente ese límite primero en los servidores con un único worker. Se observa una importante mejora en node con tres instancias para la cantidad de carga que estamos aplicando. Hay que tener en cuenta la cantidad de usuarios que va a consumir este servicio.

Observando la mediana de los gráficos (de tiempo de respuesta) muestra números más bajos, los números máximos son significativamente más altos (aunque sean pocos los request que lleguen a tener ese tiempo de respuesta), lo cual tambien podria aportar a tener una mala experiencia en cuanto a la User-Perceived Performance.

Si se desea tener un tiempo de respuesta específico (o máximo) se podría utilizar alguna técnica de detección. Una es el Ping/Echo, donde un componente realiza el ping y espera que le devuelva un *echo* en ese tiempo predefinido.

Proxy

Para este endpoint, se realizaron tres tipos de simulaciones para cada servidor. Primero se hizo una simulación que crea un request cada 3 segundos de manera constante durante un tiempo. Luego, otra simulación, crea un request cada 5 segundos. Y por último, otra simulación genera una rampa, es decir, cada segundo va creando más request hasta llegar a una cantidad determinada.

A continuación se describirá y mostrará la ejecución de distintos escenarios de prueba realizados para ver cómo se comporta el endpoint proxy bajo distintos deploys.

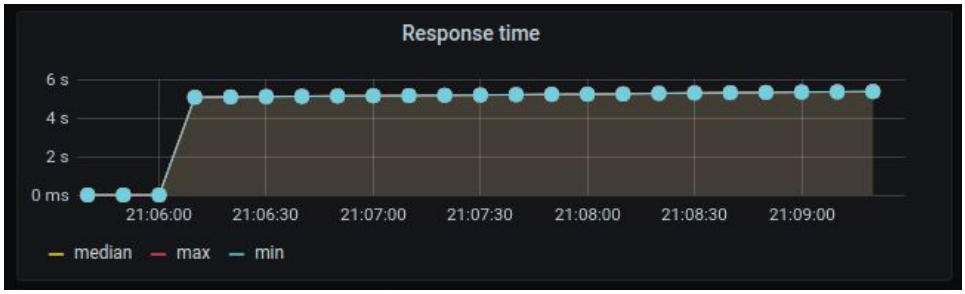
Gunicorn

Comenzando por el webserver hecho con Gunicorn (1 worker) se muestra la ejecución de dos escenarios distintos, uno donde llega un usuario cada 3 segundos, y otro donde llega un usuario cada 5 segundos. Se probaron varios escenarios pero se eligieron dos para mostrar y así ver un caso donde el web server funciona bien y un caso en donde funciona mal.

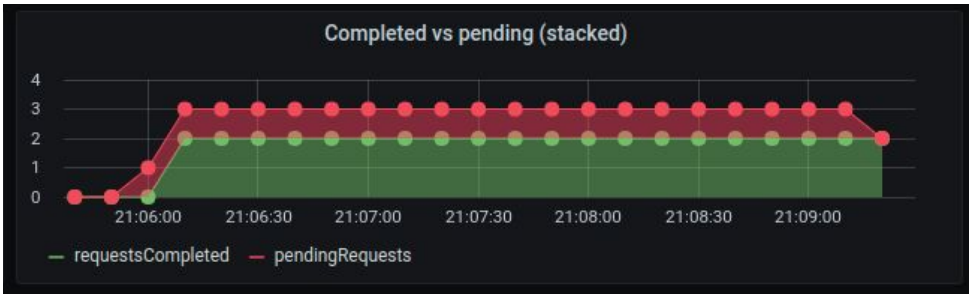
Un request cada 5 segundos

Comenzamos por el escenario en el que llega un usuario cada 5 segundos. Debido a que 5 segundos es el tiempo que dura el endpoint proxy, el hacer un request cada 5 segundos o más, permite recibir siempre una respuesta y nunca alcanzar un timeout. El endpoint proxy simula ser un endpoint en el cual se realiza una acción y se espera un resultado, como por ejemplo acceder a una base de datos o hacer un request a un endpoint. Al utilizar un web server hecho con Python, ocurre que todo se ejecuta de manera sincrónica, y por lo tanto se tiene que esperar a recibir el resultado del request o de la base de datos, para luego poder continuar.

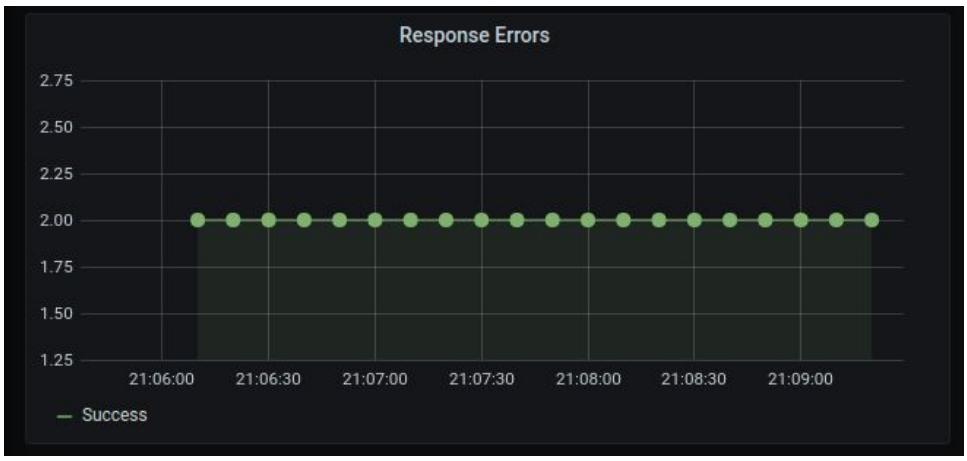
En los gráficos se puede observar un tiempo de respuesta constante de 5s. El mismo no varía porque llega un request cada 5s, con lo cual cada uno puede terminar antes de que llegue otro. También podemos ver que todos los requests realizados son exitosos y no hay errores por timeout.



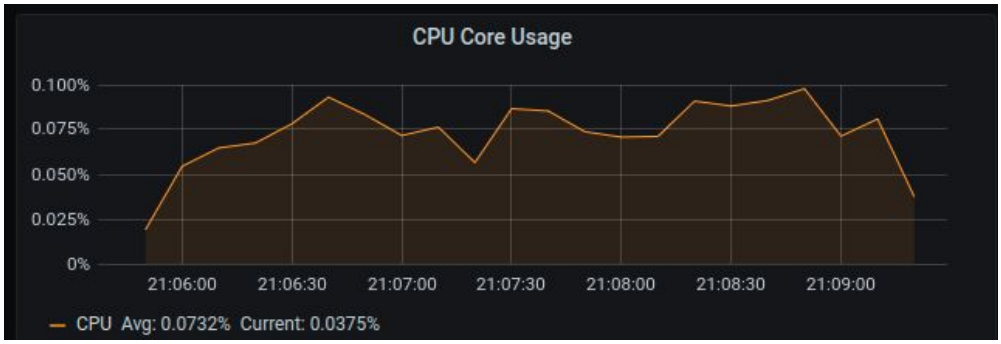
Tiempo de respuesta



Request completados vs request pendientes



Request exitosos

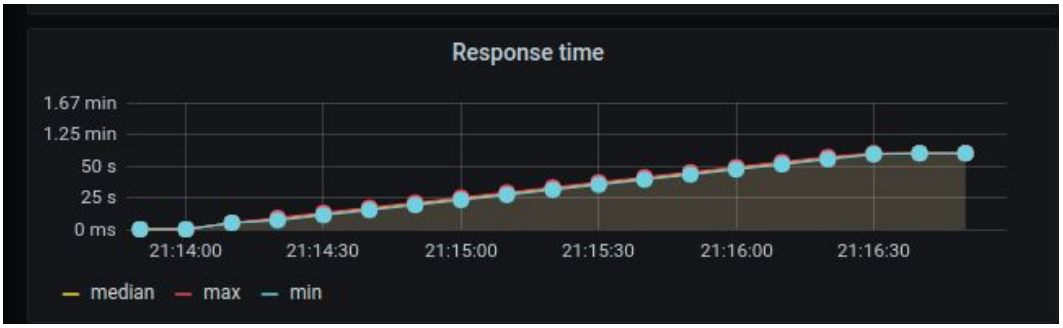


Uso de CPU

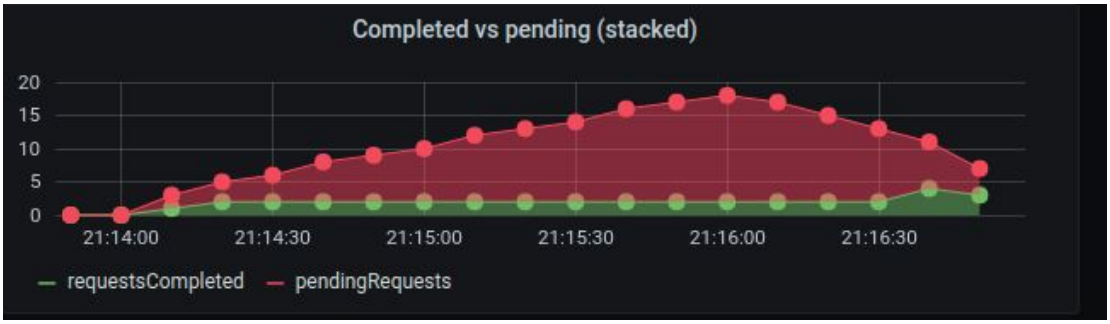
Un request cada 3 segundos

A continuación se muestran los gráficos de la ejecución con request cada 3 segundos. En este caso sí se acumulan los request (se van “encolando”). Esto se debe a que el tiempo entre request es menor a 5 segundos. Lo que ocurre es que el tiempo de espera se acumula y el tiempo de respuesta aumenta de manera lineal indefinidamente, por lo tanto siempre llega un punto en el que se acumulo tanto tiempo que se supera el timeout (en este caso el timeout es de 1 min) y tenemos error en el servidor y el request no se puede atender.

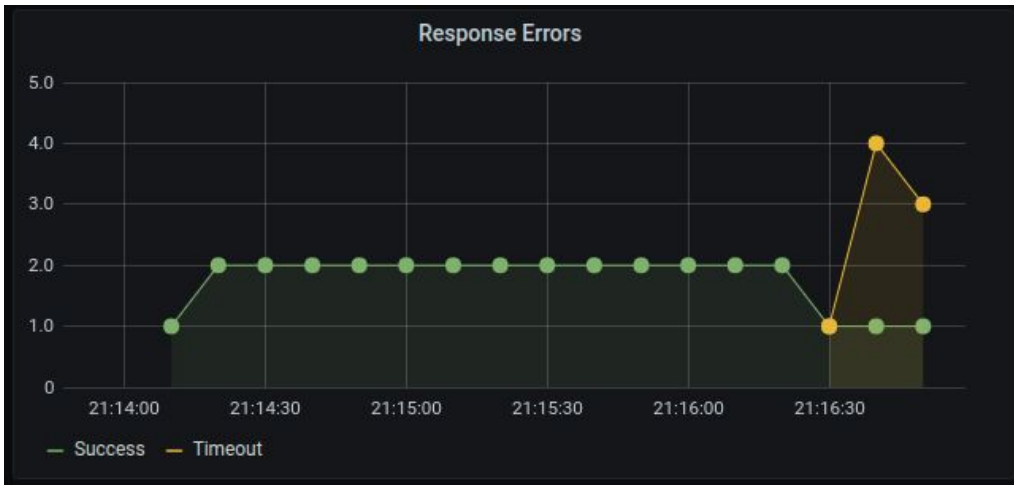
Se puede ver que el response time aumenta de manera lineal hasta llegar a una constante que es el timeout del servidor, una vez que el tiempo de respuesta lo supera, se empieza a tener errores por timeout.



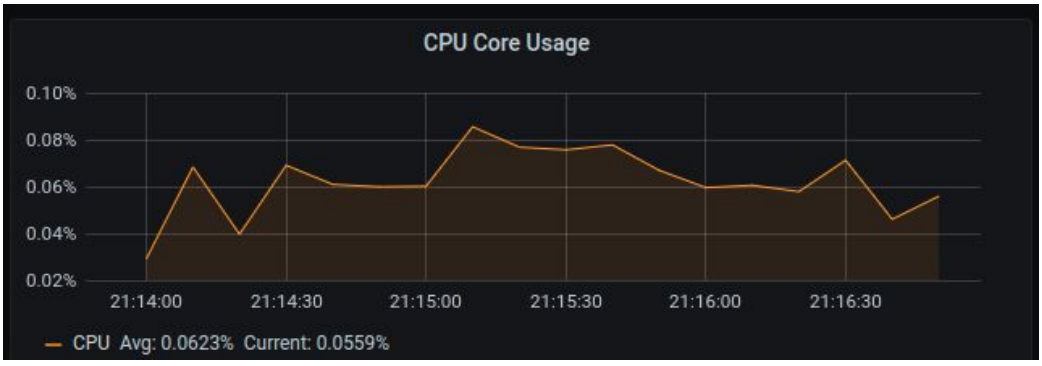
Tiempo de respuesta



Pendientes y completados



Respuestas de requests

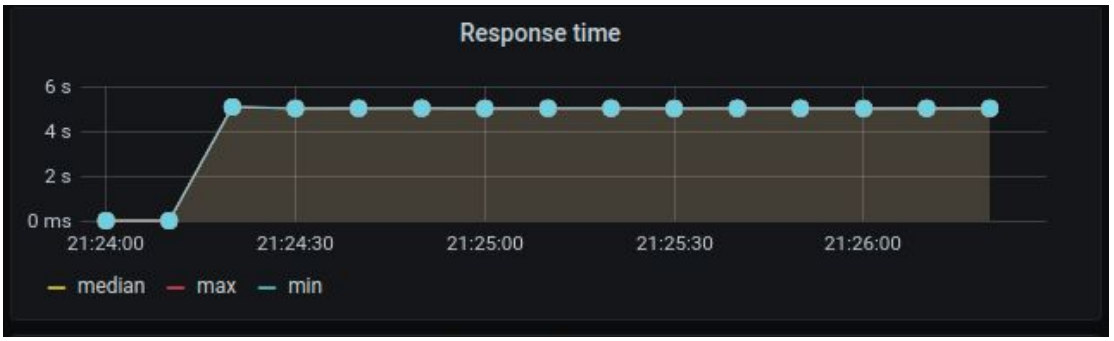


Uso del CPU

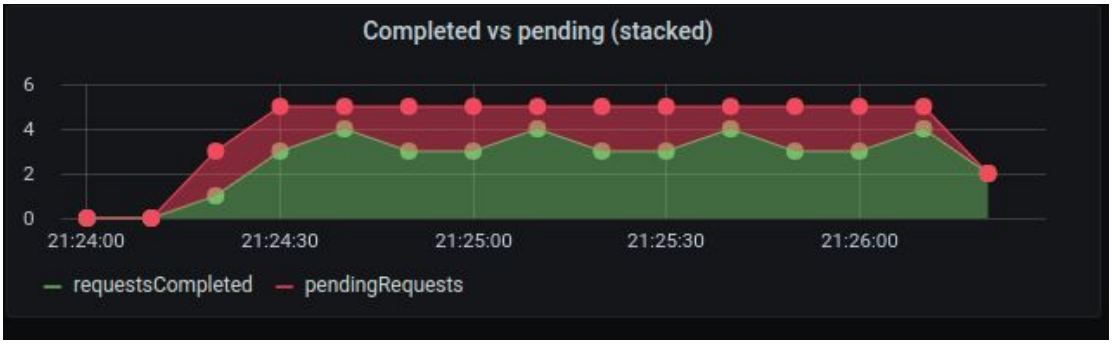
Node

Continuando con el servidor hecho con Express.js (framework de Node), se decidió no mostrar los gráficos de la simulación de un request cada 5 segundas ya que no se pudieron ver datos significativos que valga la pena comparar y se vió que la simulación de un request cada 3 segundos ya mostraba resultados en los que no se ve ningún error por timeout. Lo ocurre es que al ser asincrónico Node Js se da cuenta de que puede resolver ciertas tareas de manera asincrónica y aprovechar mejor el tiempo. En este caso, se da cuenta de que no tiene sentido esperar por un sleep y continua atendiendo otros requests de manera concurrente para luego volver a los mismos cuando sea necesario. El sleep simula ser por ejemplo un acceso a una base de datos o un request a una api externa. Por lo tanto, con NodeJs podemos recibir muchos request proxy sin ningún problema, mucho más de un request cada 3 segundos.

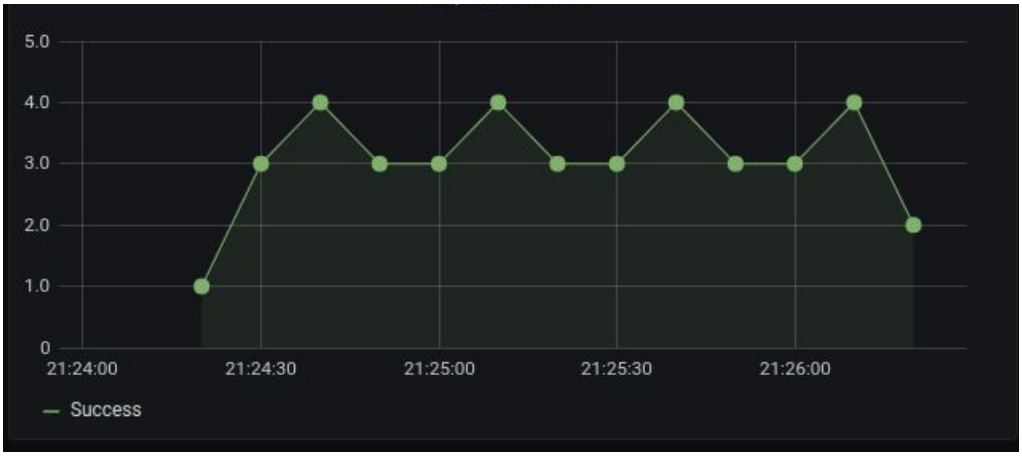
Un request cada 3 segundos



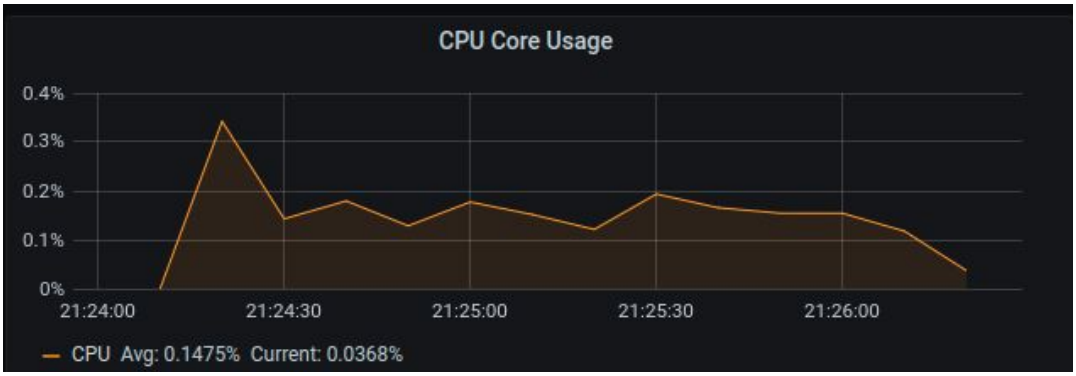
Tiempo de respuesta



Pendientes y completados



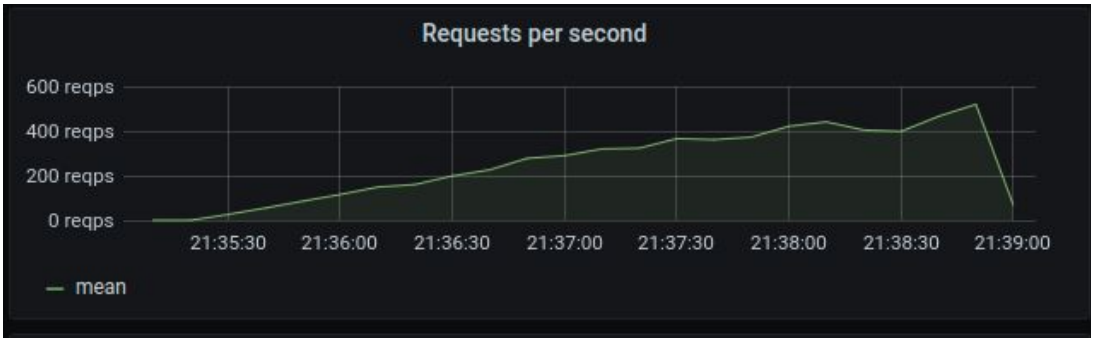
Respuestas de requests



Uso del CPU

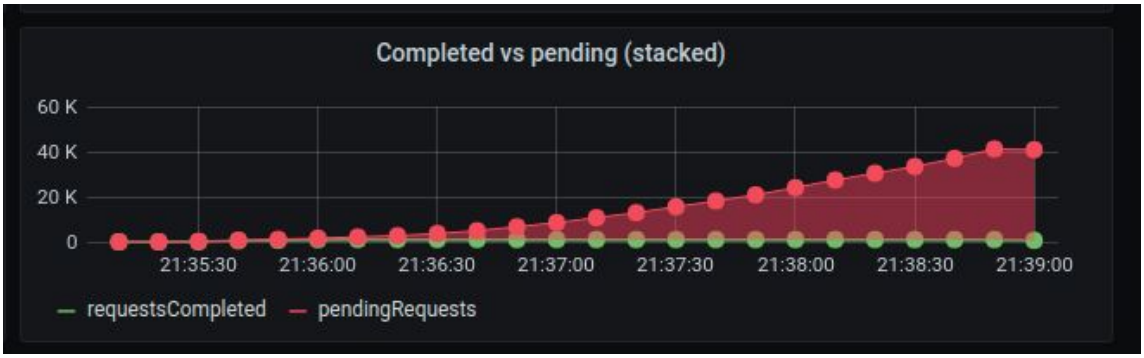
Rampa

Para este escenario se debe tener en cuenta que, al tener un worker del load balancer, no se pueden tener más de 1024 conexiones, obteniendo el error ECONNRESET al intentar crear una nueva conexión (si se pasa de 1024 conexiones). Esto tiene que ver con que cada proceso en linux, no puede tener más de 1024 file descriptor, por lo tanto no podemos tener más de 1024 conexiones TCP abiertas al mismo tiempo, y ocurre entonces que se empiezan a rechazar conexiones TCP cuando el server supera este número. Para resolver esto se podría aumentar el número de workers y con esto también las workers connections.

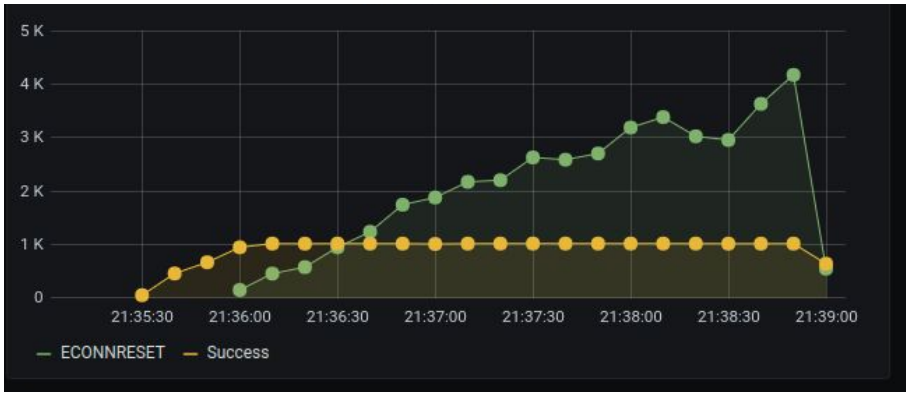


Requests por segundo

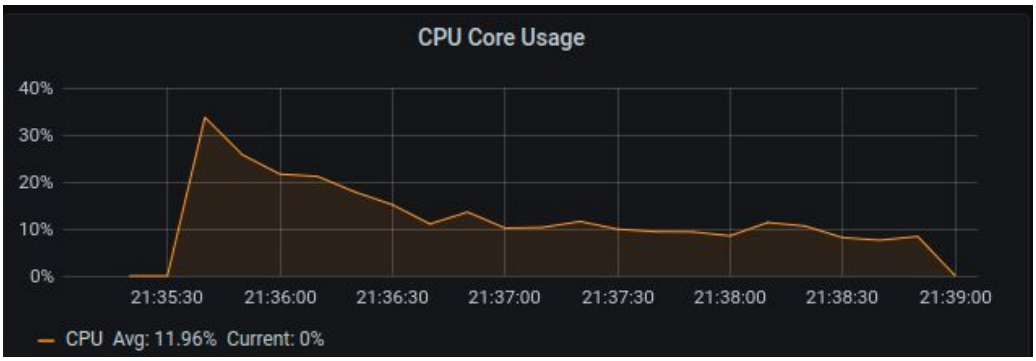
Tiempo de respuesta



Pendientes y completados



Respuestas de requests

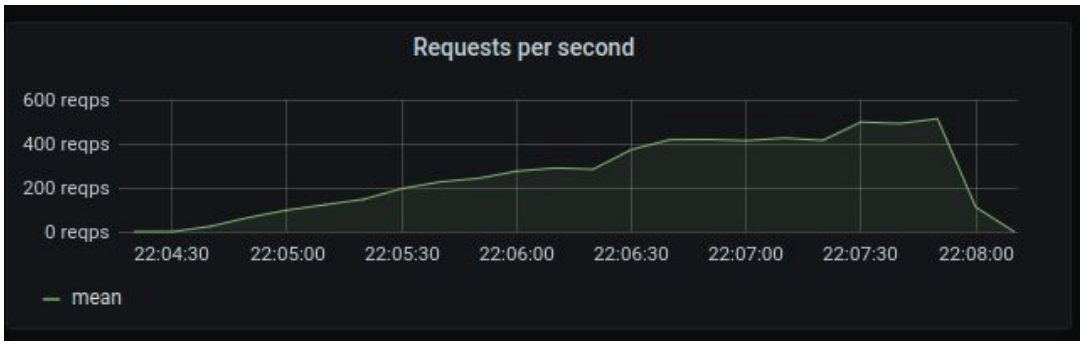


Uso de CPU

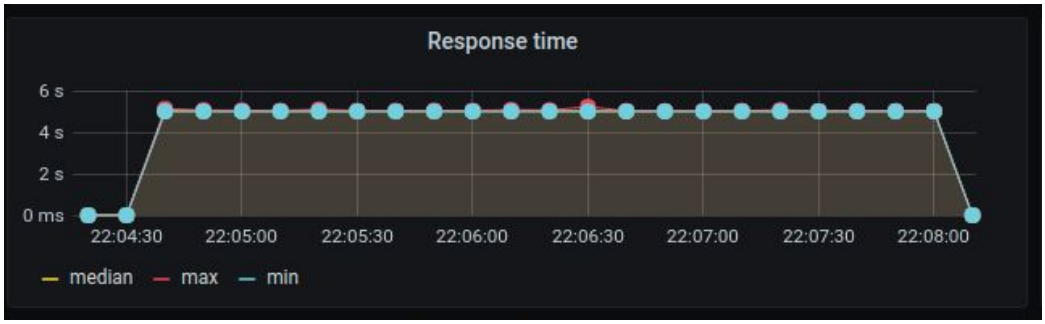
Node Replicado

Rampa

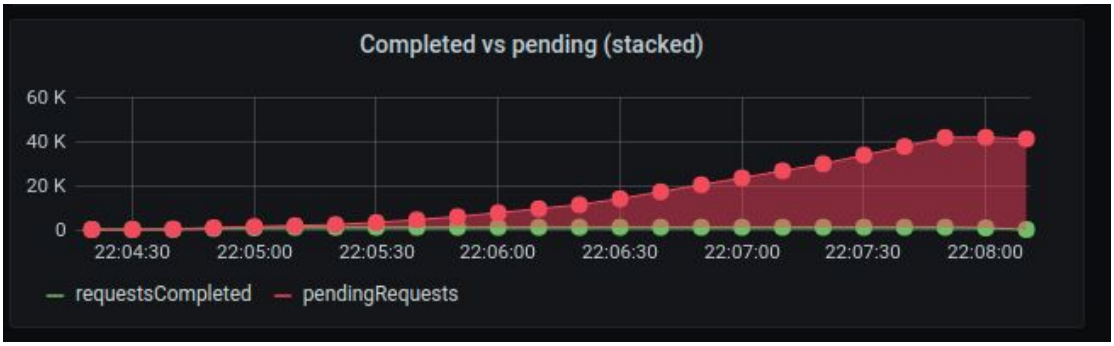
Como se mencionó anteriormente, como se alcanza el máximo de conexiones TCP (1024), no se logra ver una clara diferencia entre Node con 1 aplicación y Node replicado.



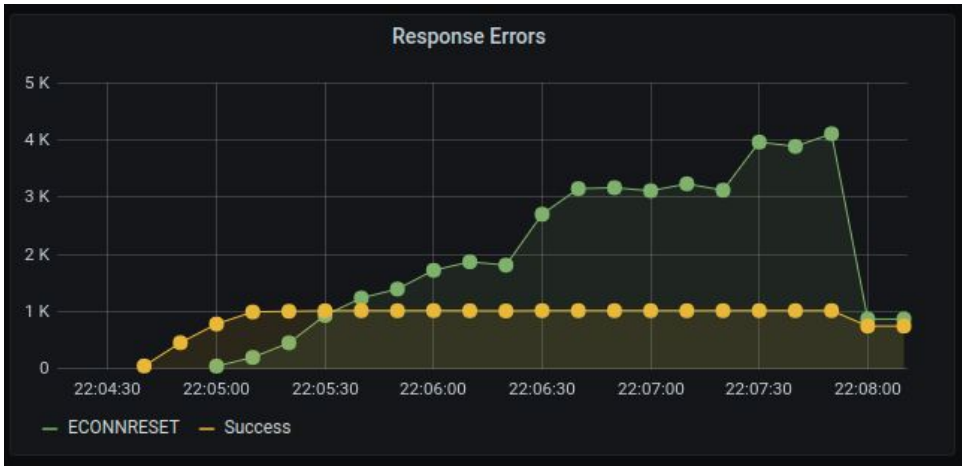
Request por segundo



Tiempo de respuesta

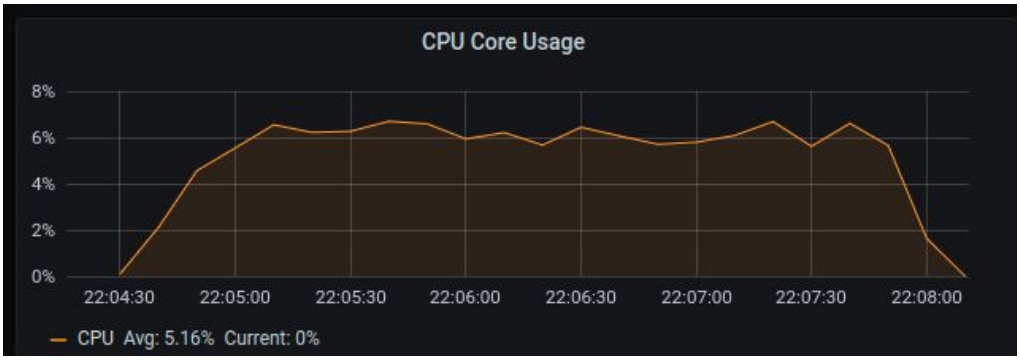


Pendientes y completados

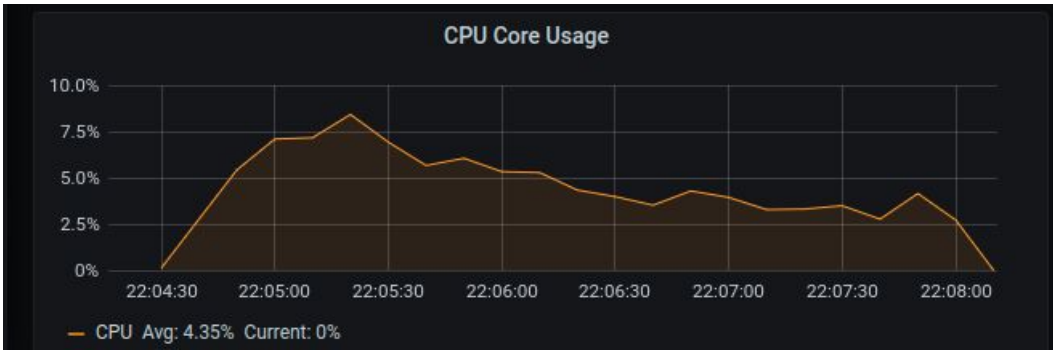


Respuestas de requests

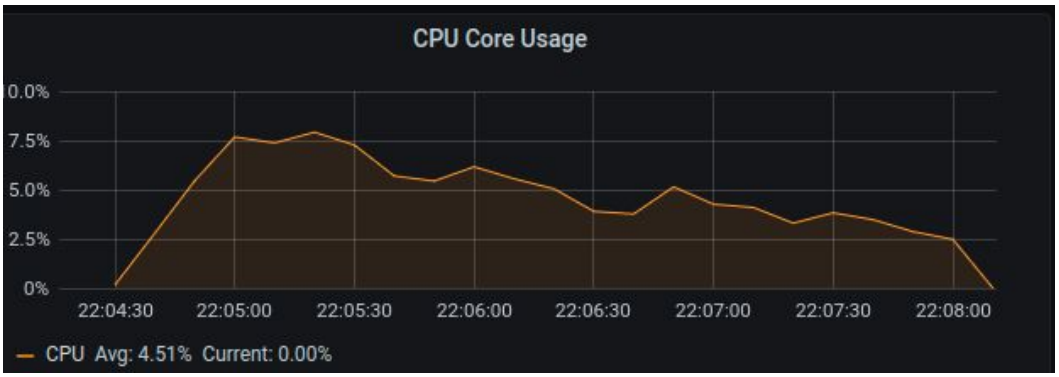
CPU Nginx



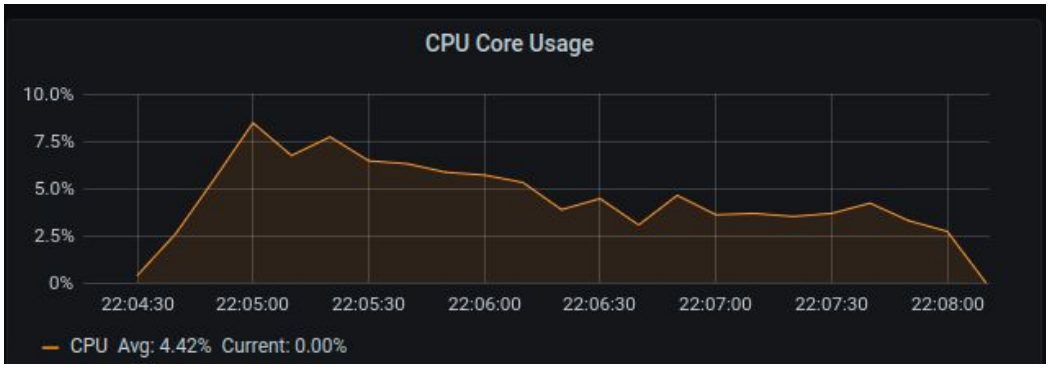
CPU Node 1



CPU Node 2



CPU Node 3



Intensive

A continuación se describirá y mostrará la ejecución de distintos escenarios de prueba realizados para ver cómo se comporta el endpoint intensive bajo distintos deploys.

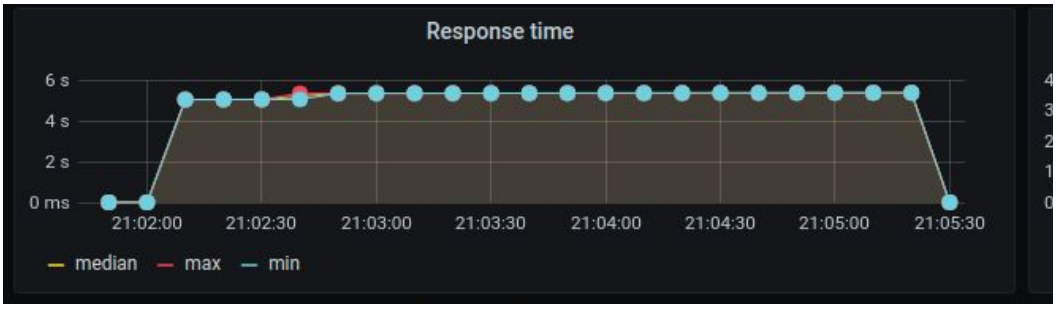
Gunicorn

Comenzando por el web server hecho con Gunicorn (1 worker) se muestra la ejecución de dos escenarios distintos, uno donde llega un usuario cada 3 segundos, y otro donde llega un usuario cada 5 segundos. Se probaron varios escenarios pero se eligieron dos para mostrar y así ver un caso donde el web server funciona bien y un caso en donde funciona mal.

Un request cada 5 segundos

Comenzamos por el escenario en el que llega un usuario cada 5 segundos. Debido a que 5 segundos es el tiempo que dura la tarea intensiva, el hacer un request cada 5 segundos o más, permite recibir siempre una respuesta y nunca alcanzar un timeout.

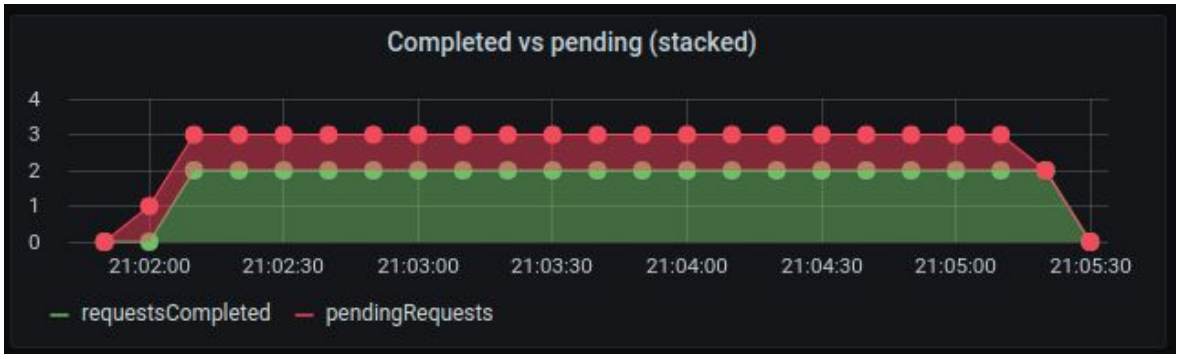
En los gráficos se puede observar un tiempo de respuesta constante de 5s. El mismo no varía porque llega un request cada 5s, con lo cual cada uno puede terminar antes de que llegue otro. También podemos ver que todos los request realizados son exitosos y no hay errores por timeout.



Tiempo de respuesta

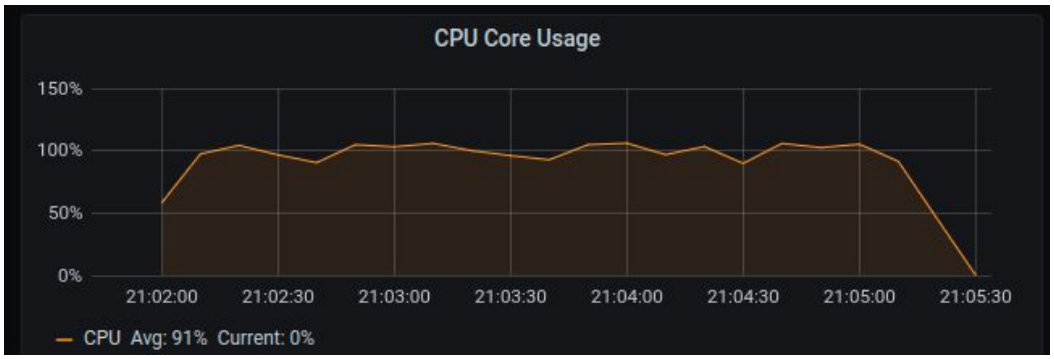
Request exitosos

También podemos ver una cantidad constante de request completados y pendientes en cada reporte. Esto, otra vez, se debe a que los request no se acumulan debido al tiempo que hay entre los mismos.



Request completados vs request pendientes

Se puede observar el uso elevado del CPU. Esto se debe a que es un endpoint intensivo, y que simula hacer procesamiento.

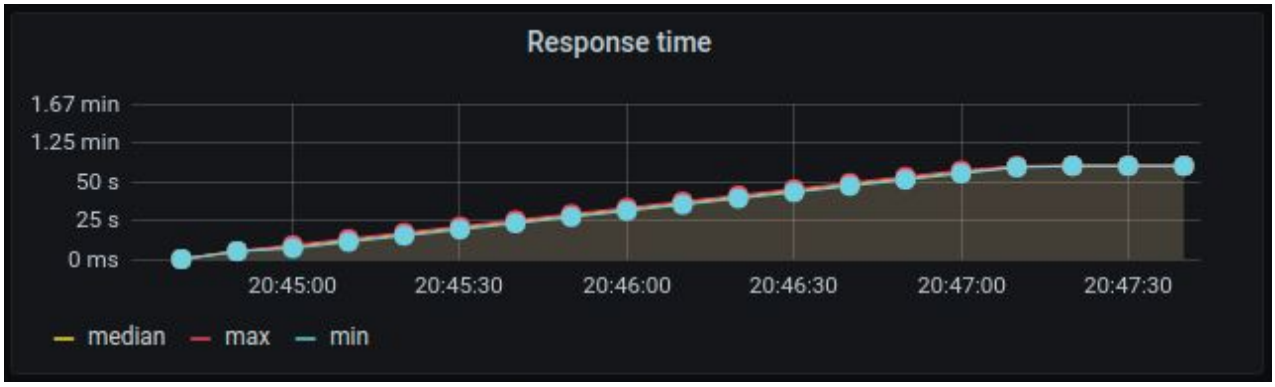


Uso de CPU

Un request cada 3 segundos

A continuación se muestran los gráficos de la ejecución con request cada 3 segundos. En este caso sí se acumulan los request (se van “encolando”). Esto se debe a que el tiempo entre request es menor a 5 segundos. Lo que ocurre es que el tiempo de espera se acumula y el tiempo de respuesta aumenta de manera lineal indefinidamente, por lo tanto siempre llega un punto en el que se acumulo tanto tiempo que se supera el timeout (en este caso el timeout es de 1 min) y tenemos error en el servidor y el request no se puede atender.

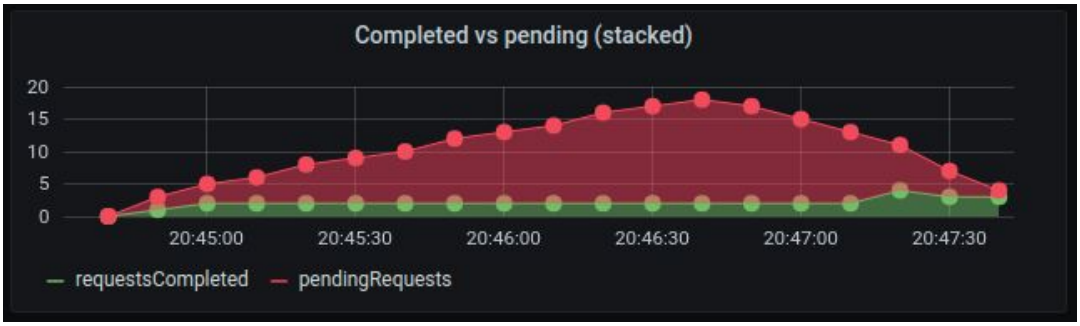
Se puede ver que el response time aumenta de manera lineal hasta llegar a una constante que es el timeout del servidor, una vez que el tiempo de respuesta lo supera, se empieza a tener errores por timeout.



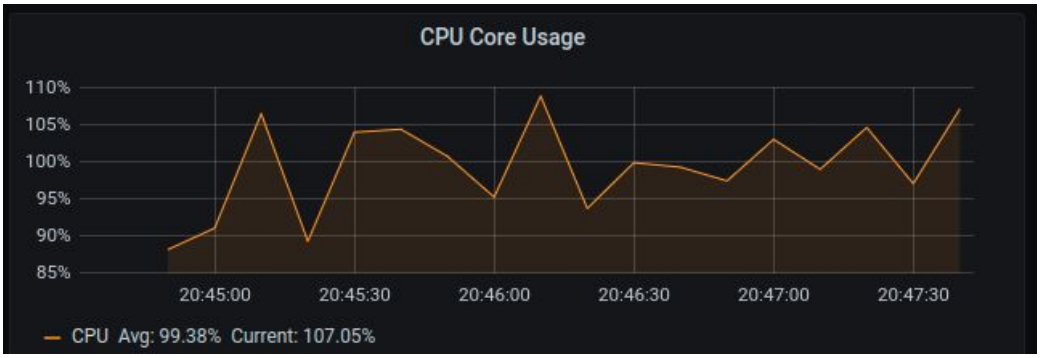
Tiempo de respuesta

Request exitosos y request con timeout

Se puede ver que la cantidad de request pendientes aumenta de manera lineal, esto está relacionado al aumento lineal en el tiempo de respuesta. Luego disminuye la cantidad de request pendientes ya que el server deja de recibir request y los que tenía que resolver terminaron exitosamente o con un timeout.



Request completados y pendientes

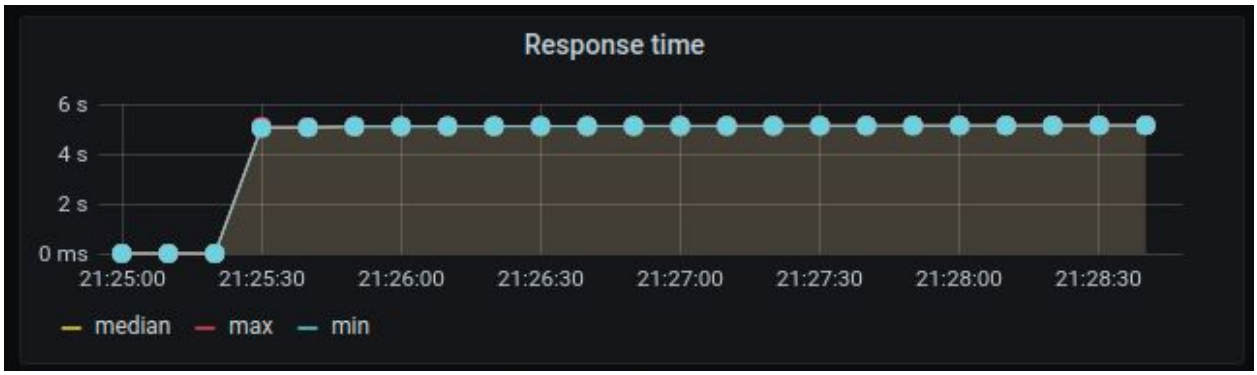


Uso de CPU

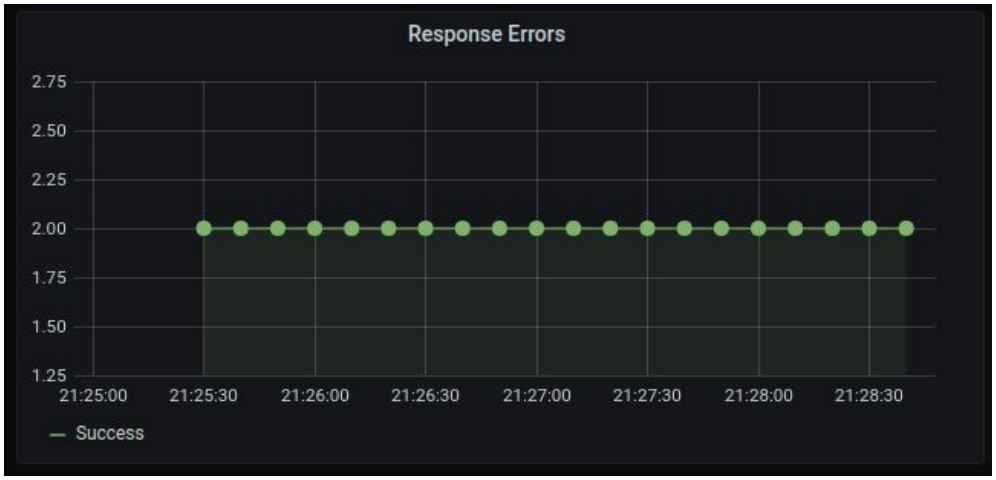
Node

Continuando con el servidor hecho con Express.js (framework de Node), se obtienen gráficos idénticos. Si bien NodeJs permite el asincronismo, debido a que la tarea del endpoint es intensiva, no se aprovecha el asincronismo y se tienen que esperar que termine la ejecución de un request para poder atender esto.

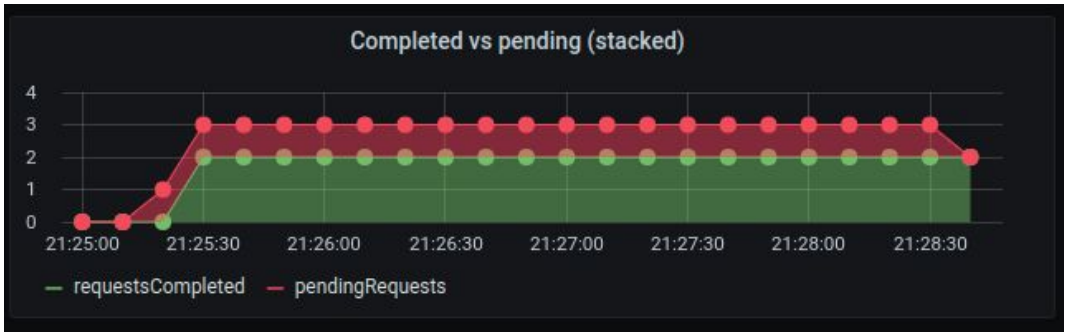
Un request cada 5 segundos



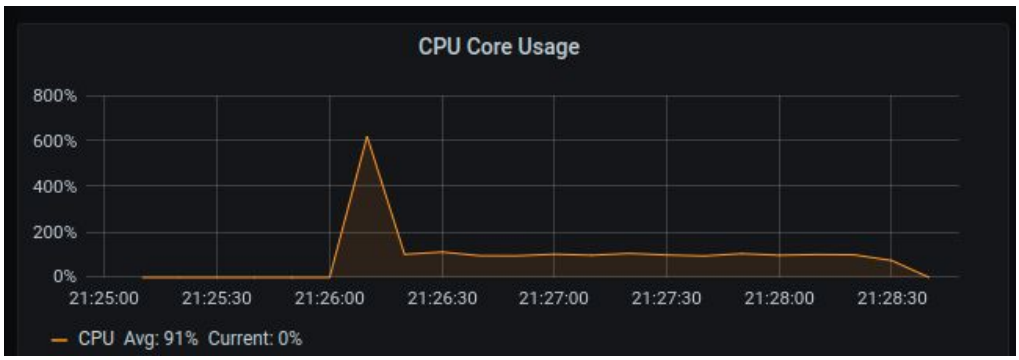
Tiempo de respuesta



Request exitosos

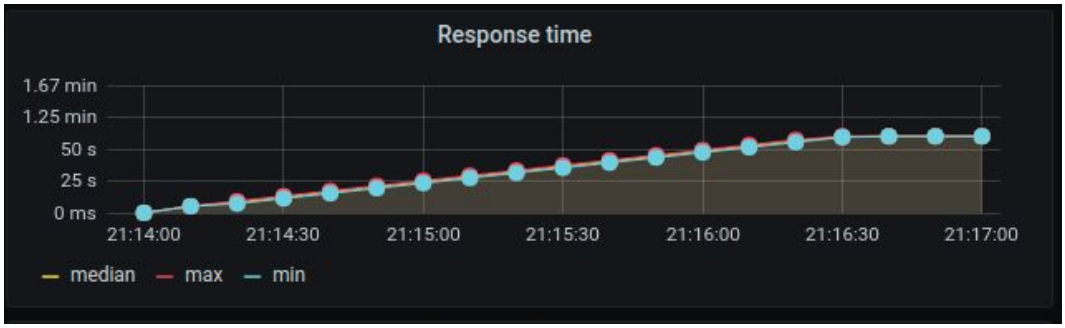


Request completados vs request pendientes

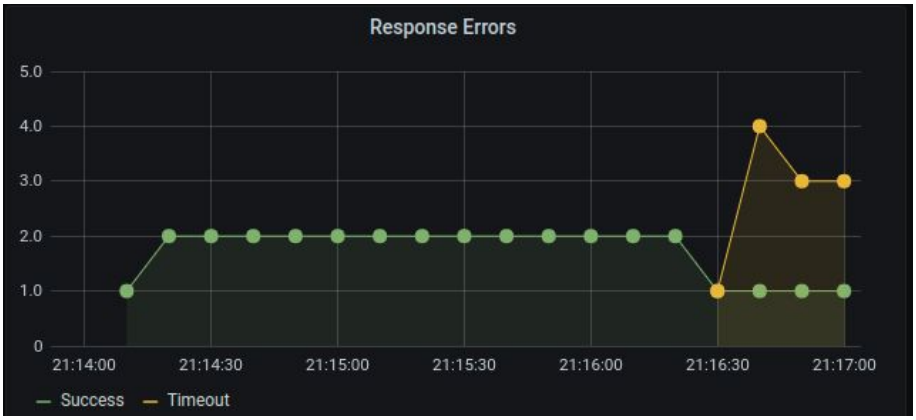


Uso de CPU

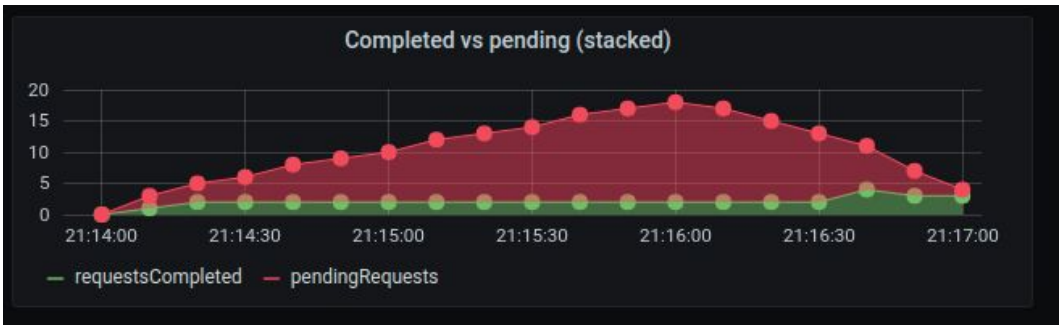
Un request cada 3 segundos



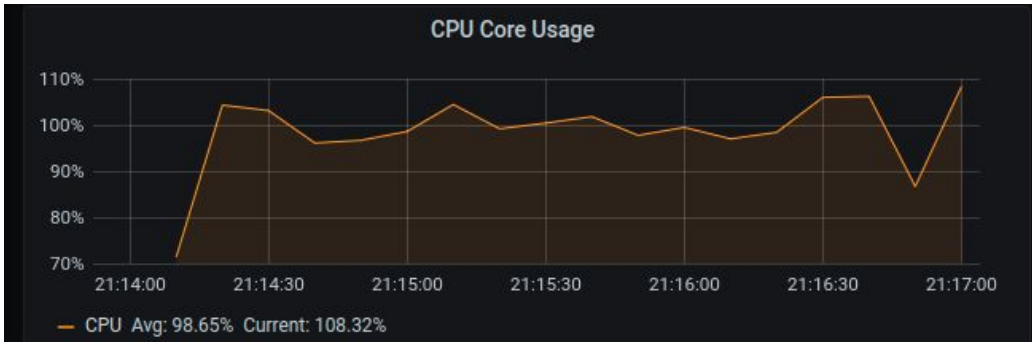
Tiempo de respuesta



Request exitosos y request con timeout



Request completados y pendientes



Uso de CPU

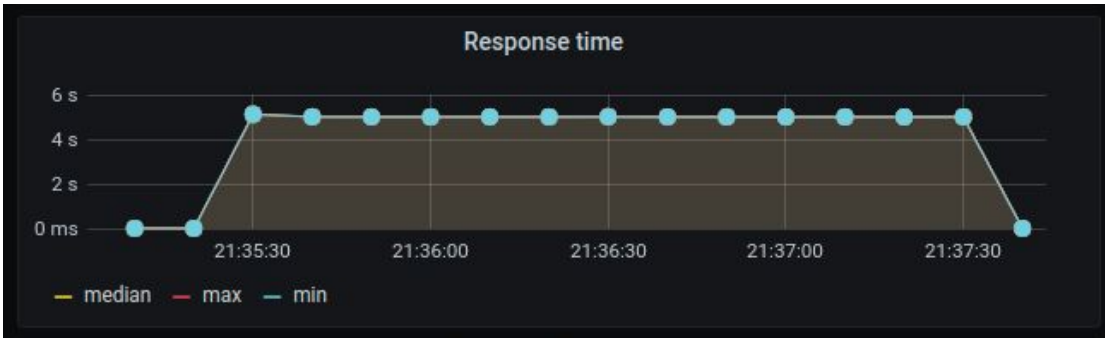
Node Replicado

Un request cada 3 segundos

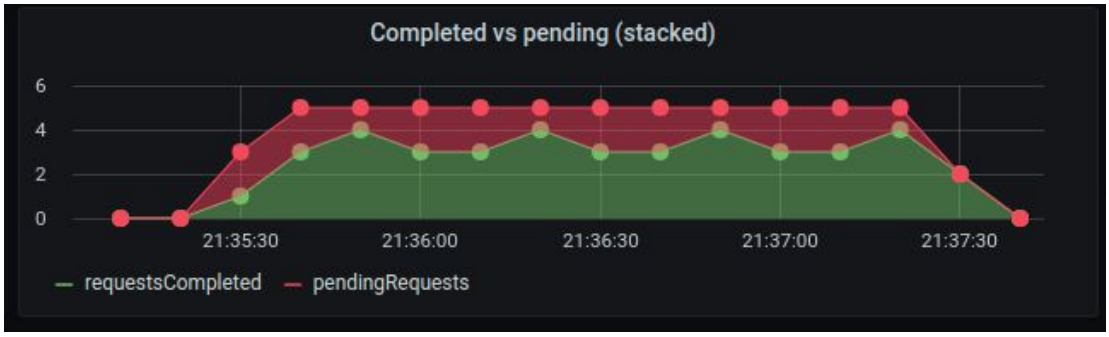
Para el caso del servidor Node replicado en tres containers se decidió hacer la misma prueba de un request cada 3 segundos para poder hacer una comparación con los otros casos no replicados que comenzaban a tener problemas de timeout. Lo que ocurre es que al tener 3 réplicas de la aplicación de node, si bien llegan 3 request por segundo al load balancer, a cada una de las réplicas les llega un request cada 9 segundos. Tampoco tendríamos problemas si al load balancer le llegaran request cada 2 segundos, ya que cada réplica recibiría un request cada 6 segundos, y esto sigue siendo mayor que el tiempo de respuesta de un request. Por lo tanto el replicar la aplicación nos permitió evitar la acumulacion lineal de request y poder responder a más request sin llegar a errores por timeout.

Como se puede ver en los gráficos a continuación, el tiempo de respuesta de los request se mantiene constante y es significativamente menor que los anteriores.

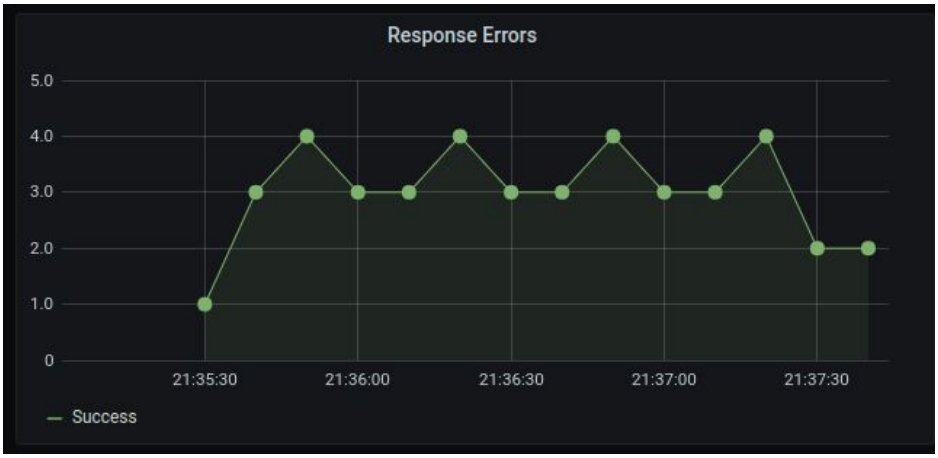
Si bien se ve que se detectaron request en estado “pendiente”, no se acumularon tanto como en los casos de los servidores no replicados, completando los request de manera pareja.



Tiempo de respuesta



Request completados y pendientes



Request exitosos

Otra medida a destacar es que el uso del CPU en cada contenedor replicado es menor al servidor no replicado, como se esperaba.

Cpu memoria Node 1

Cpu memoria node 2

Cpu memoria node 3

Impactos en atributos de calidad (Intensive)

En las simulaciones realizadas se puede decir que para el endpoint intensive la gran cantidad de request por segundo va a afectar significativamente de manera negativa la disponibilidad del servicio. Ya que, como se vió en los servidores no replicados, luego de un tiempo, el sistema se encuentra con request fallidos (por timeout), dejando de brindar un servicio consistente.

Al realizar simulaciones en el caso replicado, esto no ocurrió y se puede concluir que es una buena táctica para poder mejorar la disponibilidad del servicio (si es que se necesita tener una significativa cantidad de requests).

En cuanto al User-Perceived Performance se puede decir que (analizando el gráfico de request cada 3 segundos en los servidores no replicados) también se va a ver afectada de manera negativa ya que el tiempo de respuesta de cada pedido es cada vez mayor aunque no se llegue a una falla por timeout.

Aquí también se vio mejorado el tiempo de respuesta en el caso del servidor replicado, por lo que claramente mejora este atributo de calidad.

Se puede agregar, además, que el sistema, al tener el load balancer de nginx, es sencillo de hacerlo escalar horizontalmente agregando servidores replicados tanto para el servidor de Node como el de Python. En el caso de este último endpoint, se pudo ver que escalar de esta manera impactó positivamente.

Sección 2

Para analizar los servicios provistos por la cátedra se integró la imagen al archivo *docker-compose.yml*, con el siguiente archivo de configuración:

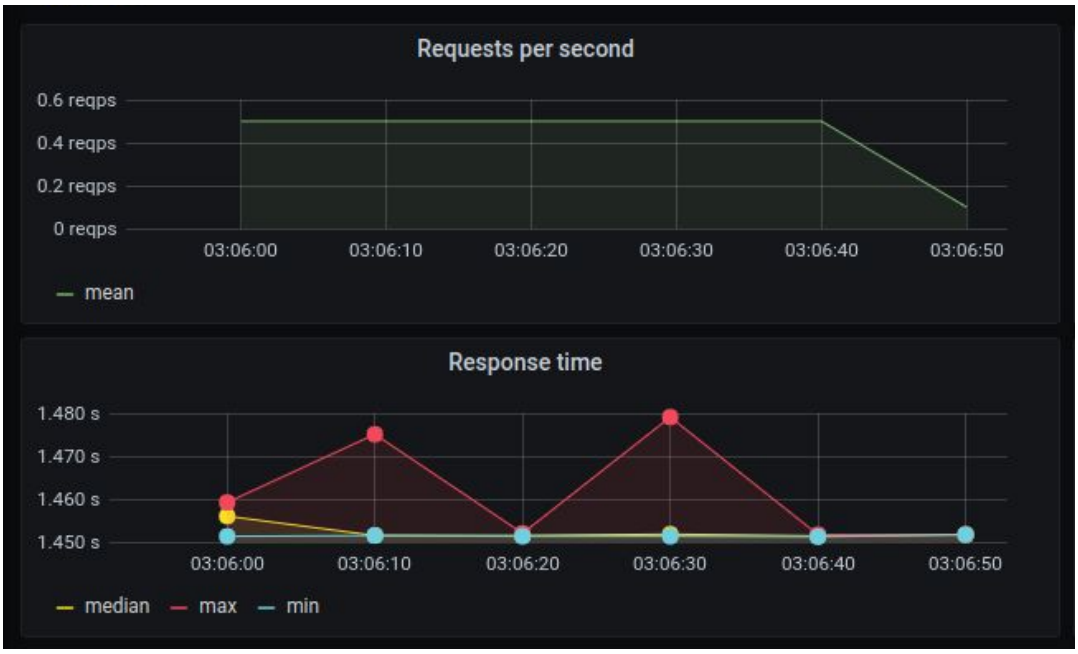
```
1 server.basePort=9090

2 group.key=bobelconstructor
```

De esta forma, el primer servidor escucha en el puerto 9090, y el segundo en el 9091.

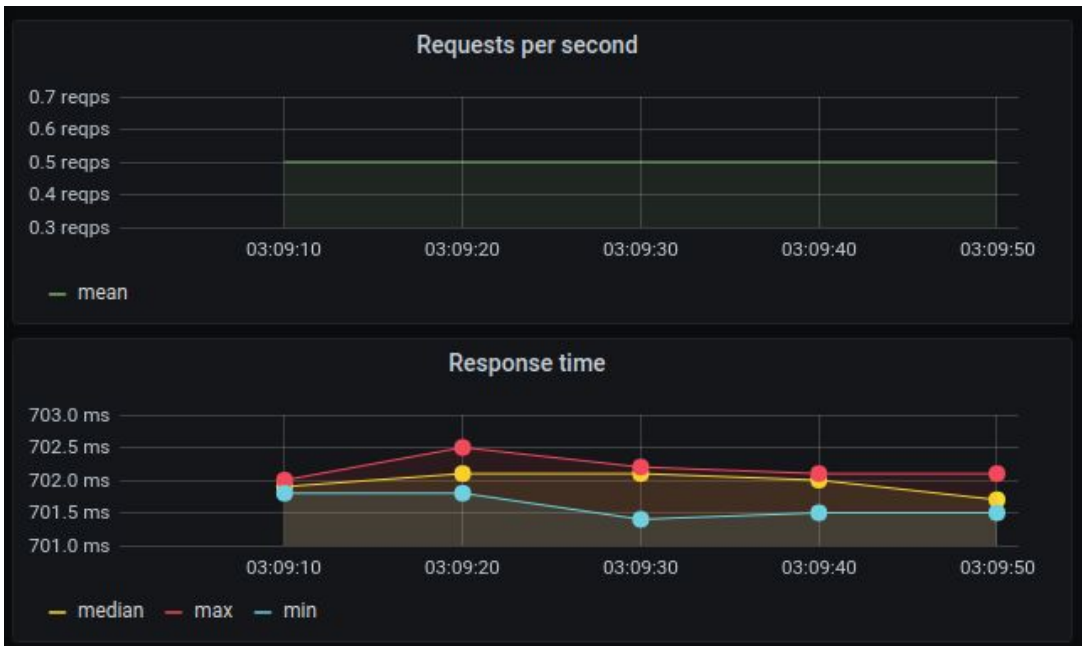
Tiempo de respuesta

Para medir el tiempo de respuesta se implementó un escenario simple en el que llega un usuario cada dos segundos, ya que simplemente se intentaba medir el tiempo de respuesta de las requests, entonces se implementó un tasa baja de arribos para darle tiempo a los servidores de procesar los pedidos uno a uno, y que los valores reportados no estén influenciados por otros pedidos que se tengan que procesar.

Para el primer servidor se obtuvo lo siguiente:



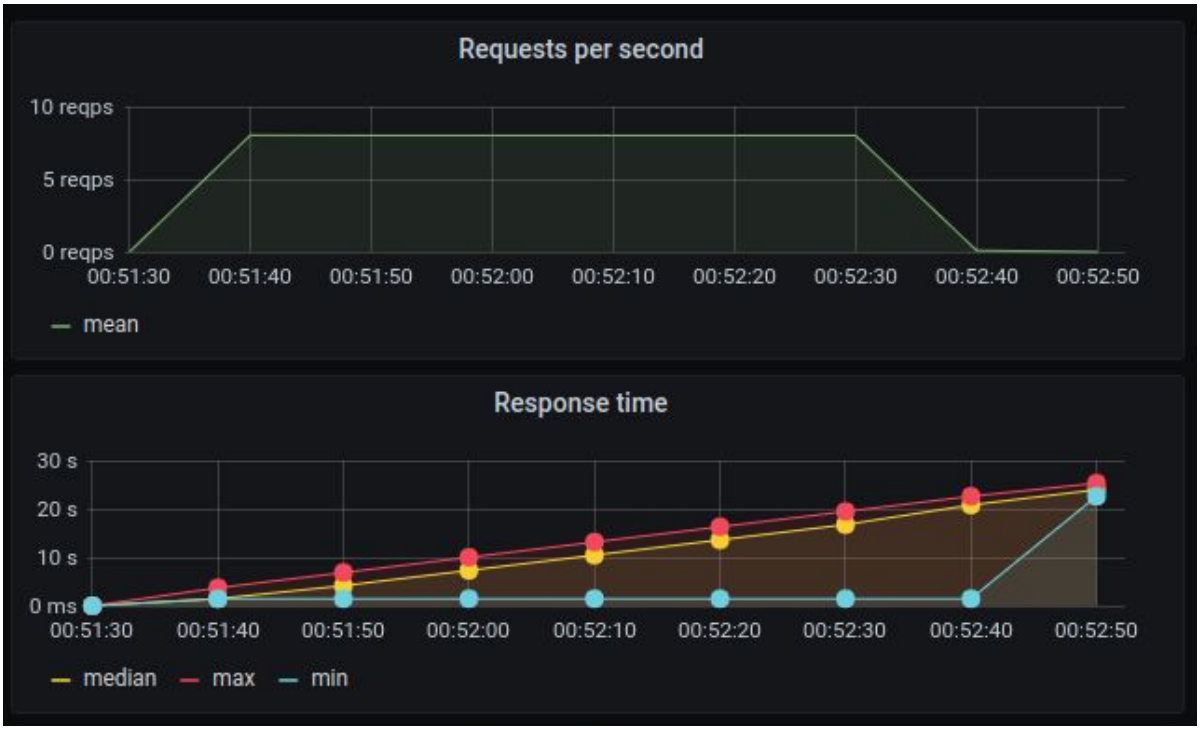
Aquí se puede ver que el tiempo de respuesta está alrededor de 1,45 segundos. Luego, con el segundo servidor se obtuvo el siguiente resultado:

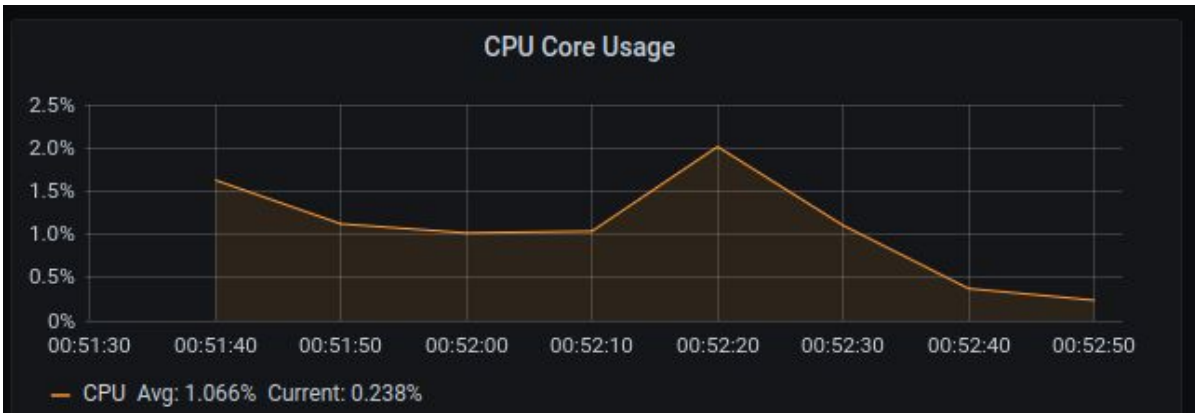


Aquí se puede ver que el tiempo de respuesta se encuentra alrededor de los 702 milisegundos.

Servidor Sincrónico o Asincrónico

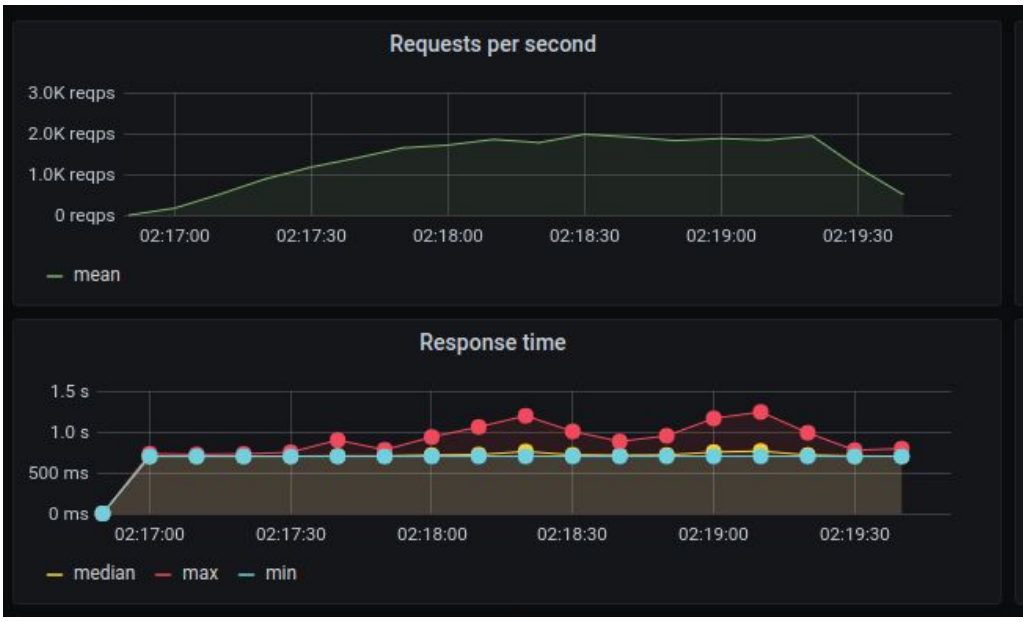
Para analizar esta característica de los servidores, se utilizó con el primer servidor un escenario simple donde llegaban 8 usuarios por segundo durante un minuto.

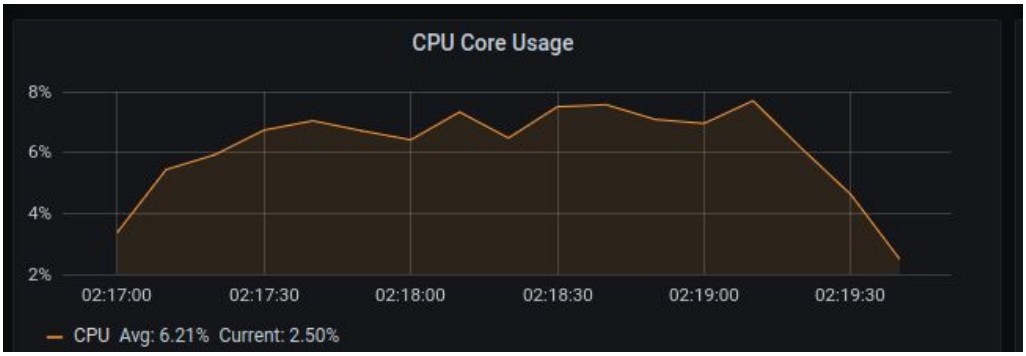




En el gráfico se puede ver que mientras se mantiene el ratio de usuarios por segundo, el tiempo de respuesta de los requests va aumentando, pero no ocurre lo mismo con el uso de CPU. Con esta información podemos decir que se trata del servidor **sincrónico**, ya que por el bajo consumo de CPU podemos decir que los requests son de procesamiento mínimo y consumen un cierto tiempo, entonces los workers, al procesar uno a la vez, por más que tengan que esperar a que termine la operación, no pueden aprovechar ese tiempo para procesar otras requests, entonces éstas se van acumulando y se van procesando luego de terminar las anteriores.

Luego, con el segundo servidor se implementó una rampa que iba aumentando paulatinamente la cantidad de usuarios por segundo y por lo tanto la cantidad de requests por segundo.



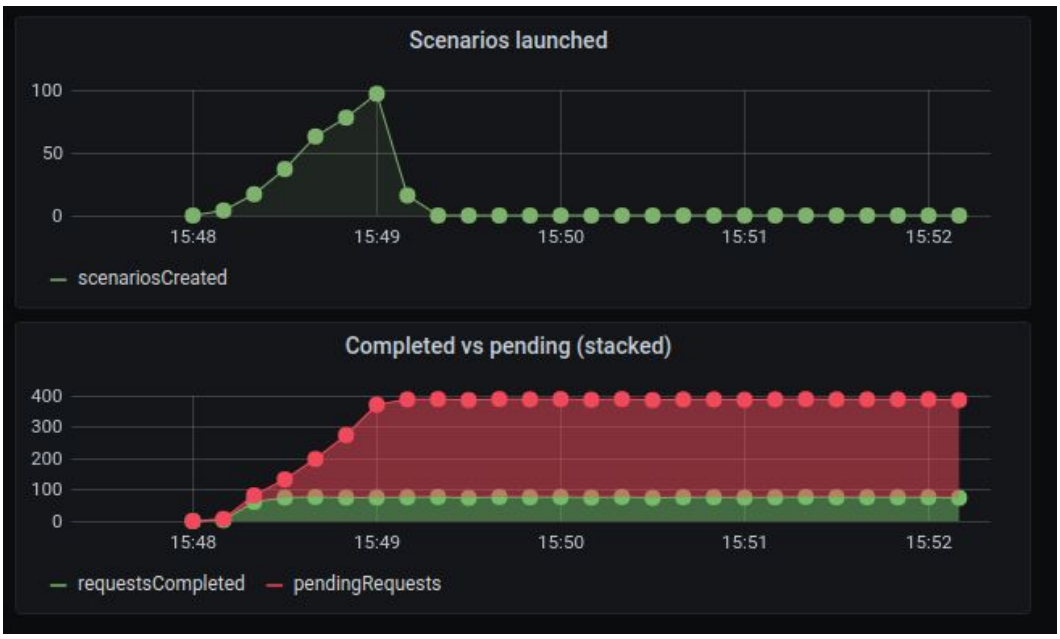


En este caso se puede ver que aunque vaya aumentando la cantidad de usuarios y de requests por segundo, se mantiene relativamente constante el tiempo de respuesta, debido a eso podemos decir que se trata del servidor **asincrónico**, ya que puede dejar a las requests de procesamiento mínimo procesándose de manera asincrónica mientras sigue atendiendo a las siguientes, por eso se tiene un tiempo de respuesta relativamente constante.

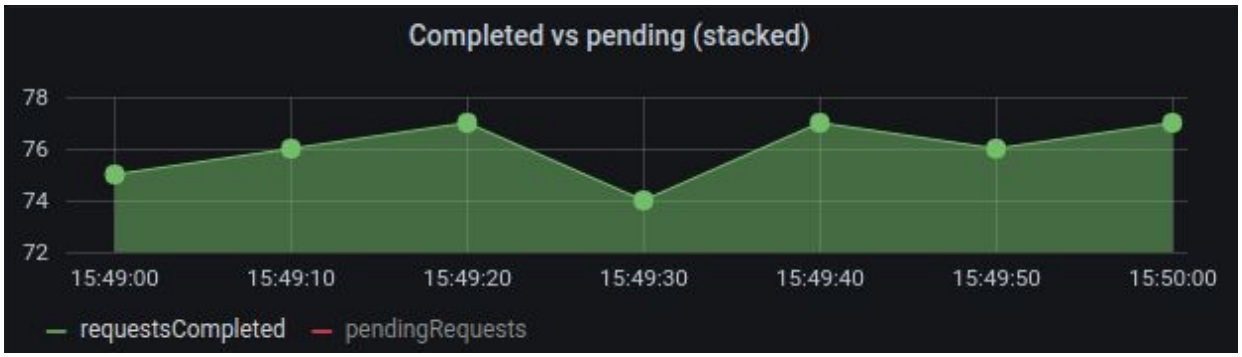
Cantidad de workers

Para medir esta cantidad, se realizó un escenario donde se enviaban al servidor sincrónico un ratio de requests muy grande hasta que se sature. Para poder observar cuántas

requests puede procesar por segundo como máximo antes de comenzar a dar errores. Los resultados fueron los siguientes:



Si hacemos un acercamiento al momento en el que la cantidad de requests completados se vuelve constante:



Podemos ver que en un momento se dejaron de lanzar escenarios en artillery, pero el servidor aún tenía los requests acumulados. Al tener tantos, los iba procesando a su máxima capacidad, la cual se ve en el gráfico que es aproximadamente 75 req/s.

Teniendo en cuenta el tiempo que dura cada request medido anteriormente, y además el tiempo en el que artillery lanza un reporte, podemos decir que un worker debería procesar en promedio 6.89 requests en 10 segundos, ya que:

*6.89 req * 1.45 s/req = 10 s*

Pero aquí vemos que se procesaron 75 requests en 10 segundos, lo cual es posible si se tienen alrededor de **11 workers**.

Resumen

	Servidor 1	Servidor 2
Tiempo de respuesta	1.45 s	702 ms
Sincrónico/Asincrónico	sincrónico	asincrónico
Cantidad de workers	11	-

Link al repositorio

Link al repositorio de código abierto: https://github.com/javier2409/arqsoft-2c2020-tp1, donde se encuentran las configuraciones iniciales para levantar el tp y los escenarios descriptos para cada uno de los endpoints.