Arquitectura de Software (75.73)
Año: 2021
TP 2: Cloud - AWS

Integrantes

Hernán de la Fuente - 95730 - hdelafuente@fi.uba.ar
Mariano Hielpos - 91244 - mhielpos@fi.uba.ar
Edson Justo - 97775 - ejusto@fi.uba.ar
Martín Picco - 99289 - mpicco@fi.uba.ar

Introducción

En el siguiente informe se mostrarán algunos análisis sobre una arquitectura montada sobre servicios de AWS (EC2, Lambdas, etc). El objetivo es variar ciertos parámetros de esta configuración para sacar conclusiones de los atributos de calidad que posee la misma.

En principio, tenemos un servidor en node montado sobre EC2, que nos permite facilmente escalar a un mayor número de instancias. El mismo ofrece tres endpoints o recursos:

- **Root**: Genera un simple response de “hello world”
- **Remote**: Se invoca un servicio externo.
- **Alternate**: Se invoca un servicio externo brindado por una Lambda
- **Cached**: Se invoca el mismo servicio externo que en Remote pero utilizando ElastiCache.

Análisis inicial

Root

Aquí buscamos entender el máximo de conexiones por segundo que soporta de manera correcta el servidor. Dado que no hay ninguna lógica de cálculo comprometida, no vemos desafío en cuanto a procesamiento sino más bien a nivel capa de tráfico se puedan mantener las conexiones necesarias.
Para ello, se corren escenarios de artillery que hagan un ramp_to a un número de RPS en específico. Luego, dejamos un tiempo determinado con peticiones planas, es decir, un mismo número de RPS.
Las métricas más importantes para detectar esto es:

- La proporción entre escenarios completados por sobre los creados para un escenario específico
- Errores percibidos por el cliente (artillery)

Remote

Cada request a este recurso deviene en otro request a un mismo servidor de python que tiene un solo worker. No es necesario lanzar diversas configuraciones de escenarios ya que está claro que eso complicará el correcto procesamiento de request del servidor de node.

Alternate

En este caso, el servidor de node invocará el servicio de la lambda que básicamente ejecuta el mismo código que la instancia de python en “Remote”. Acá la ventaja que vemos es que no tendremos el limitante de un solo worker del lado python para atender la totalidad de requests que lleguen a node. Esto podría reducir el cuello de botella aunque también tendrá su limitante en la capacidad de procesar lambdas en paralelo.

Cached

Finalmente, aquí se realiza la misma invocación que en ‘Remote’ pero con la ventaja de cachear la respuesta por medio de una KEY y así disminuir la cantidad de request que deben ir hasta la instancia de python. Una variable que nos interesará analizar es el tamaño de keys que puede tener ese caché ya que denota de alguna manera la capacidad que tendrá el mismo de hacer hits.

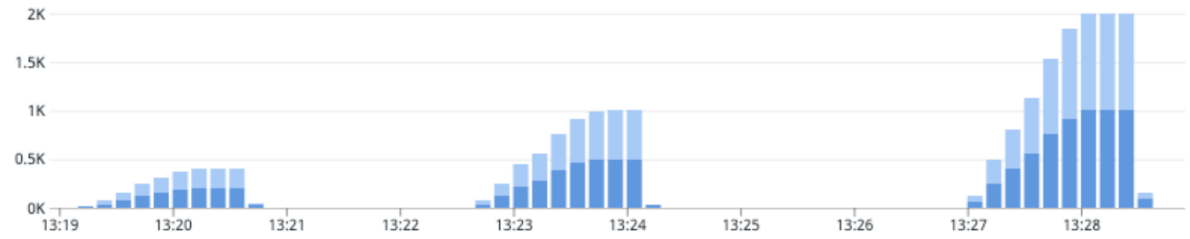
Configuración con una sola instancia

Root

Configuración de artillery con <RPS> variando entre 10 - 50 - 100

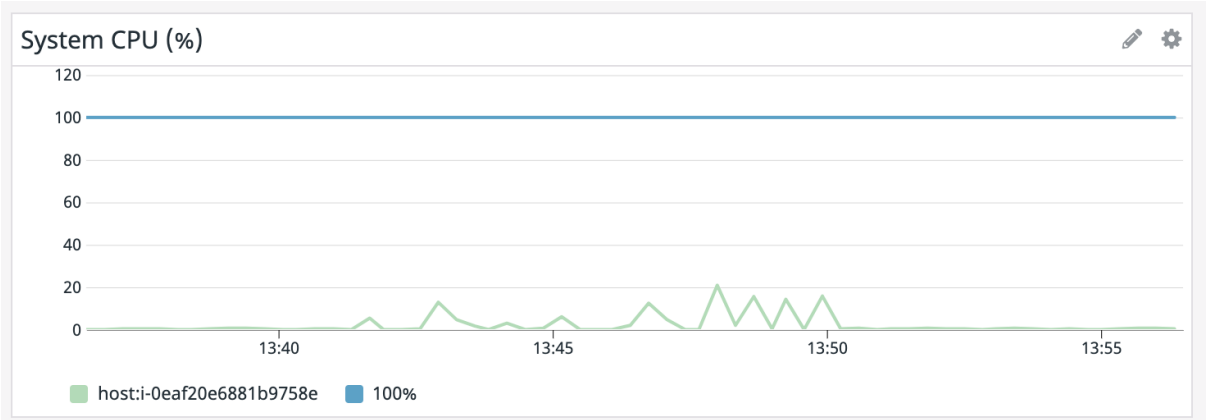
```
- name: BeginPad
    duration: 30
    arrivalRate: 0
  - name: Ramp
    duration: 60
    arrivalRate: 0
    rampTo: <RPS>
  - name: Plain
    duration: 30
    arrivalRate: <RPS>
  - name: EndPad
    duration: 30
    arrivalRate: 0
```

Comparación entre escenarios creados (request generados por el cliente) y escenarios completados (respuesta satisfactoria percibida por el cliente)



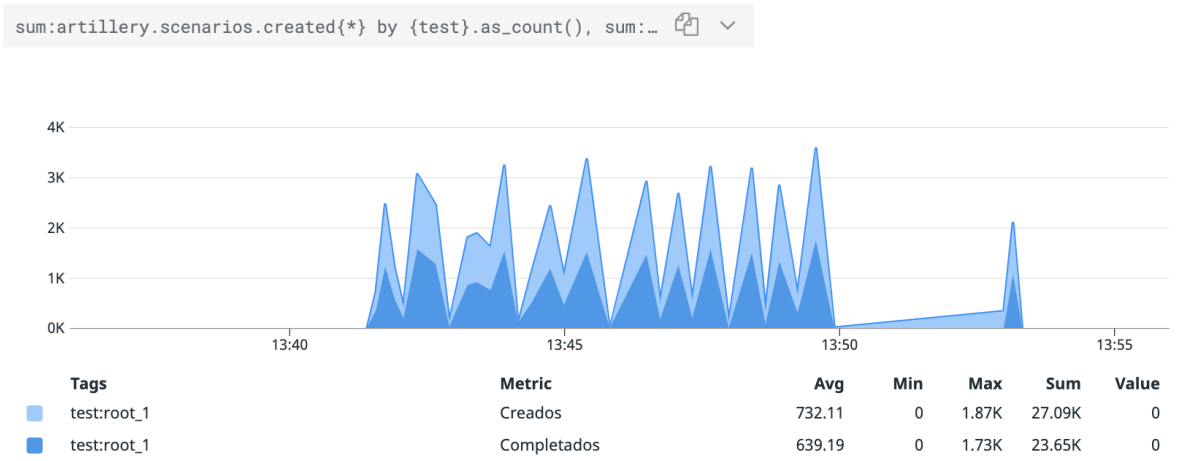
Tags	Metric	Avg	Min	Max	Sum	Value
test:root_1	Creados	238.88	0	1K	10.27K	0
test:root_1	Completados	238.88	0	1K	10.27K	0

Vemos todo correctamente. Aumentamos los RPS a 500, comenzamos a ver errores y aumentos en algunos tiempos de respuestas

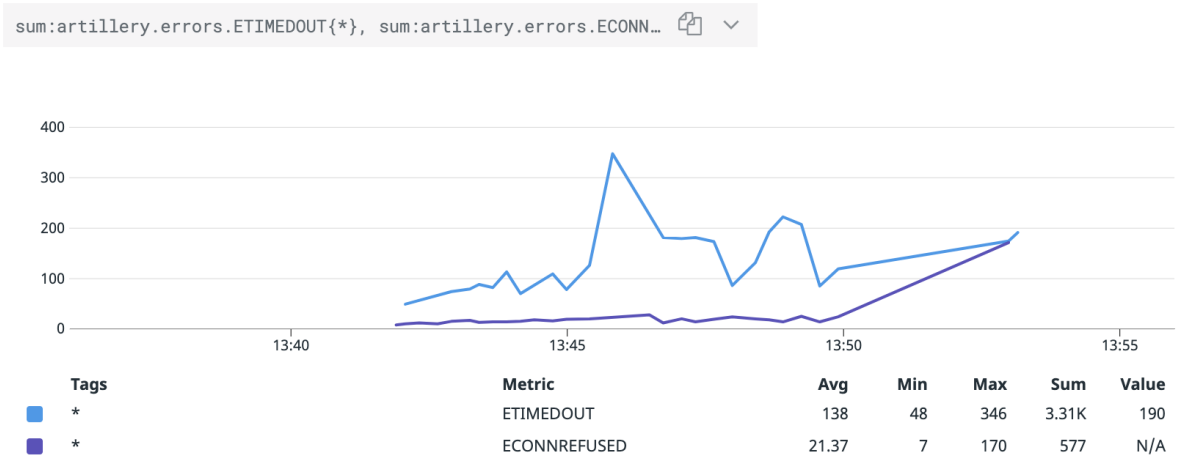

System CPU (%)
120
100
80
60
40
20
0
13:40
13:45
13:50
13:55
host:i-0eaf20e6881b9758e
100%

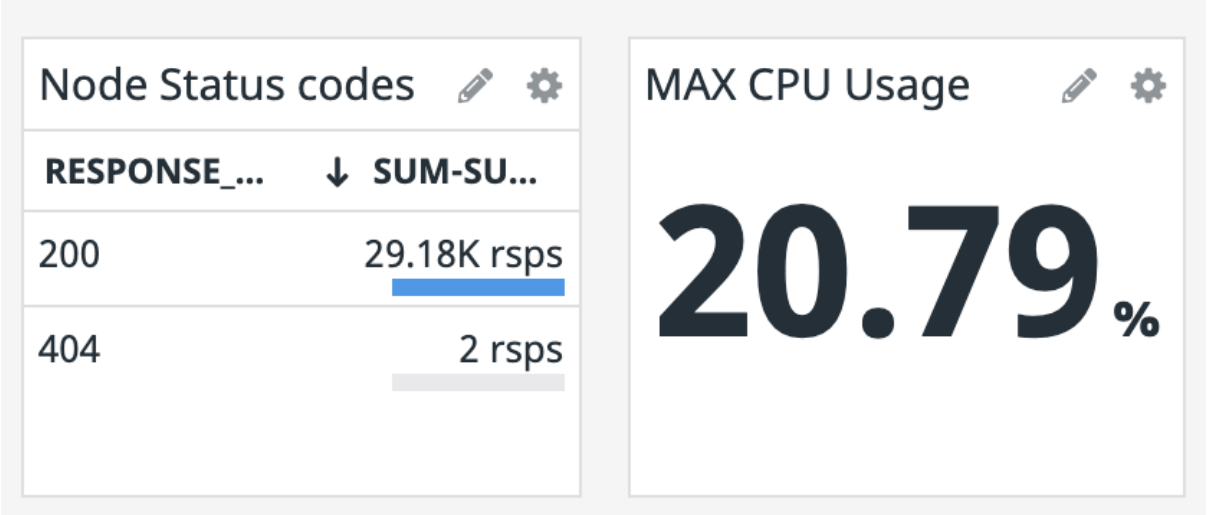

Node - Tiempos medios de respuesta {host:i-0eaf20e6881b9758e}
Seconds
15
10
5
0
13:40
13:45
13:50
13:55
i-0eaf20e6881b9758e

Comparación entre escenarios creados (request generados por el cliente) y escenarios completados (respuesta satisfactoria percibida por el cliente)



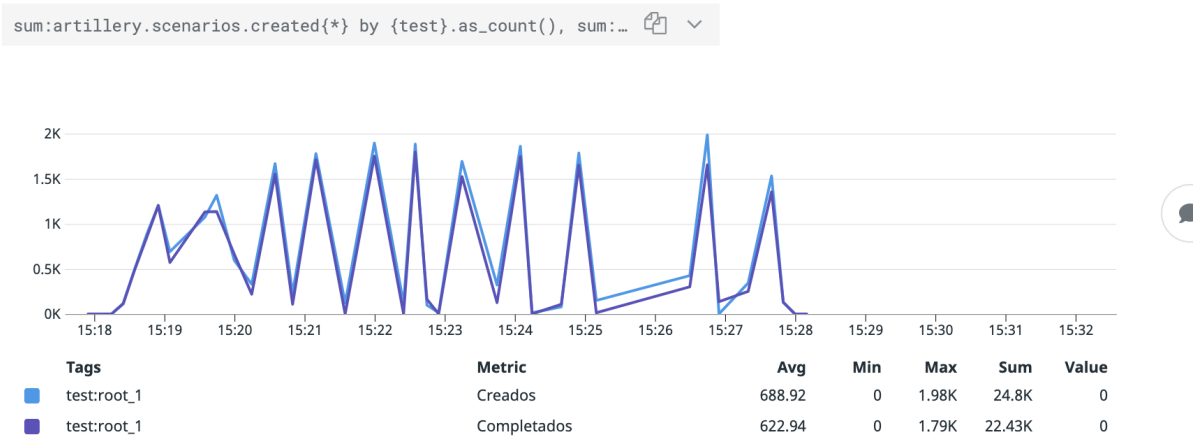
La proporción ya baja del 100% a un 87% debido a que aparecen errores como vemos en el gráfico de abajo.



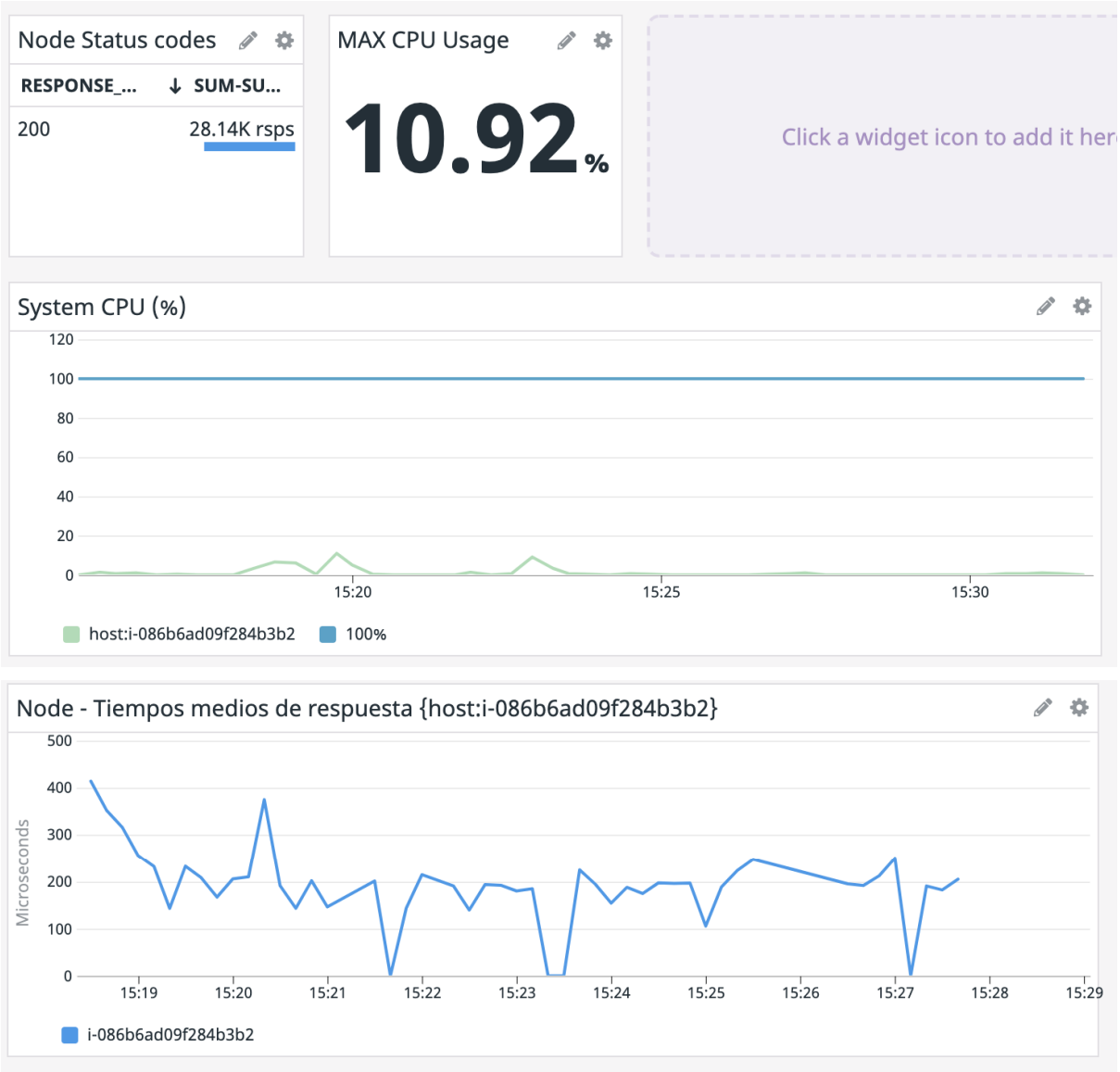


Para terminar de ver esto, optamos por una configuración más progresiva para detectar estos cambios

```
- name: BeginPad
    duration: 30
    arrivalRate: 0
  - name: Ramp
    duration: 90
    arrivalRate: 0
    rampTo: 300
  - name: Plain
    duration: 30
    arrivalRate: 500
  - name: EndPad
    duration: 30
    arrivalRate: 0
```



Aquí la proporción es de un 90%, bastante parecido al caso anterior aunque vemos un menor uso del CPU como se ve a continuación:



Conclusión

Los errores pueden ser debido a que las instancias micro no tienen la capacidad de red para poder manejar muchas conexiones en simultáneo, esto provoca que haya baches debido a que no todas los request llegan al servidor.

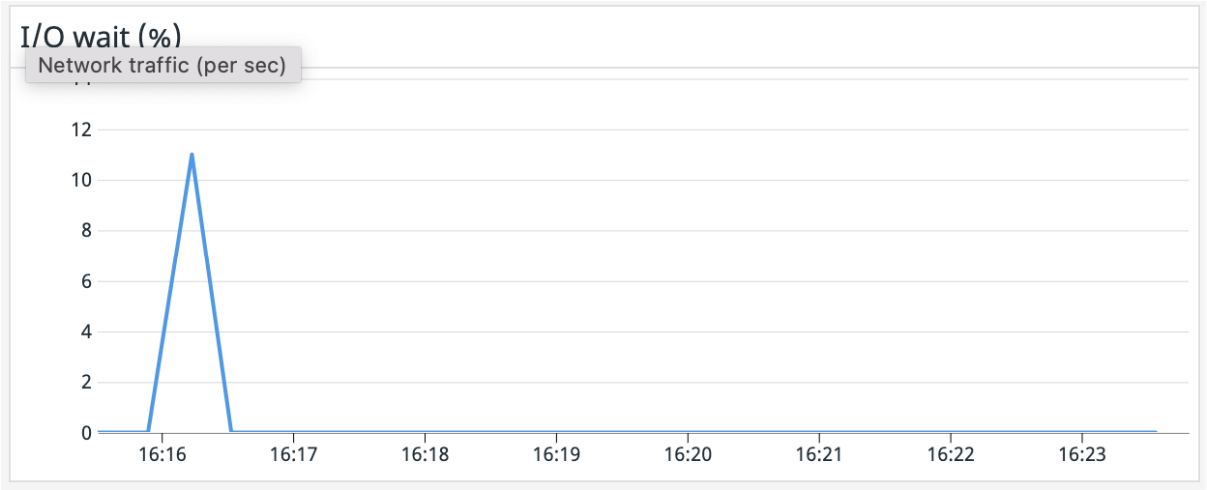
Posibles causas:

- El balanceador de carga no llega a mandar todos los request
- El servidor tiene un límite de conexiones abiertas para manejar y pasado un límite genera errores

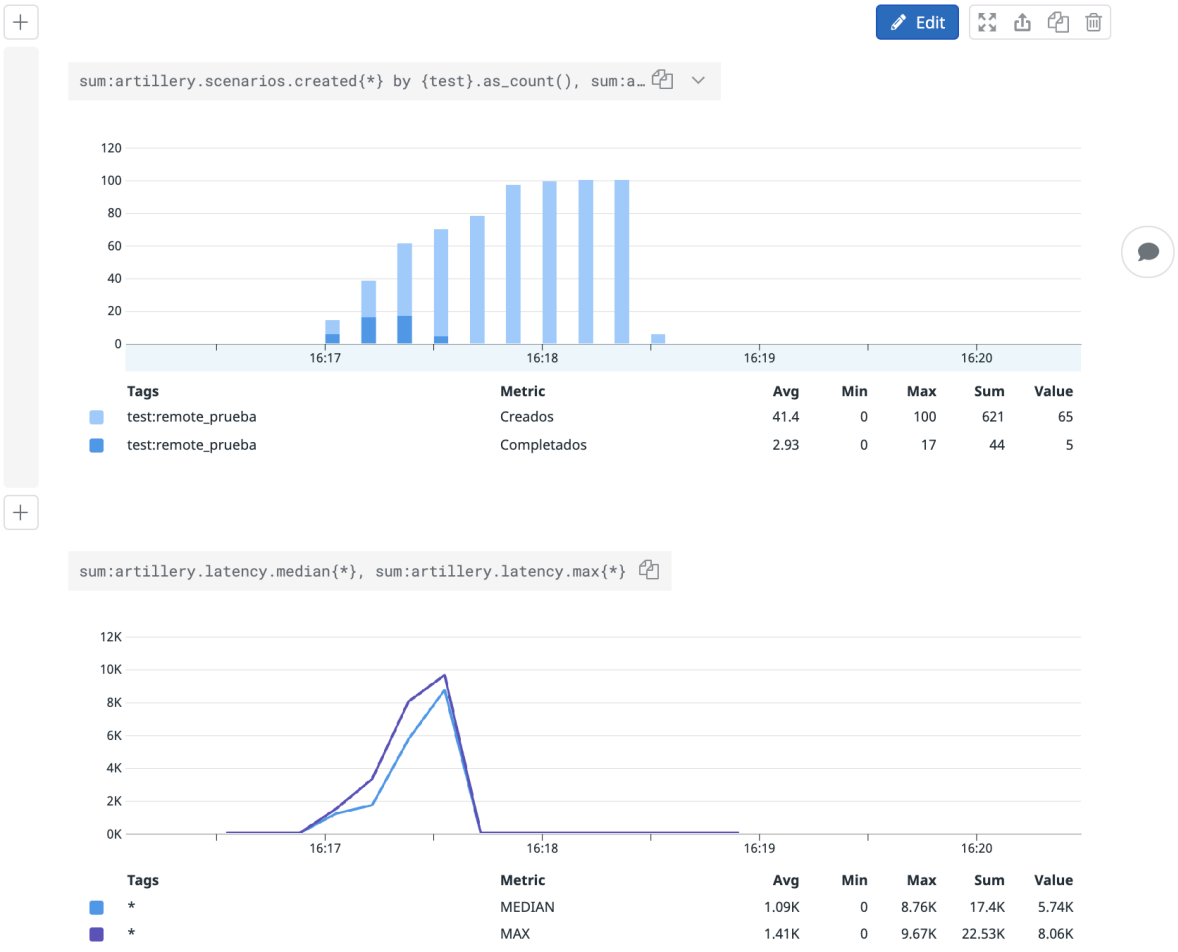
Para el correcto funcionamiento, podríamos tomar como un valor razonable unas 400 peticiones por segundo.

Remote

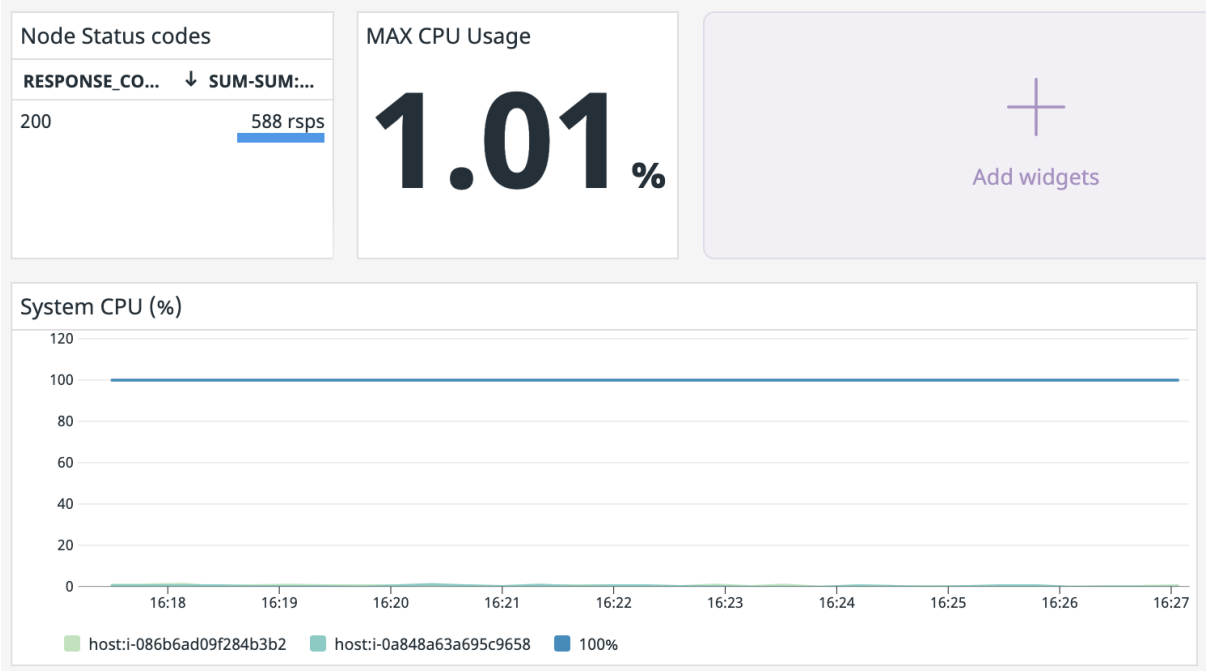
10RPS
En esta ocasión, la lectura fue mucho más sencilla. Lo primero que nos llamó la atención fue el porcentaje de IO wait que tuvo la instancia de node (~11%)

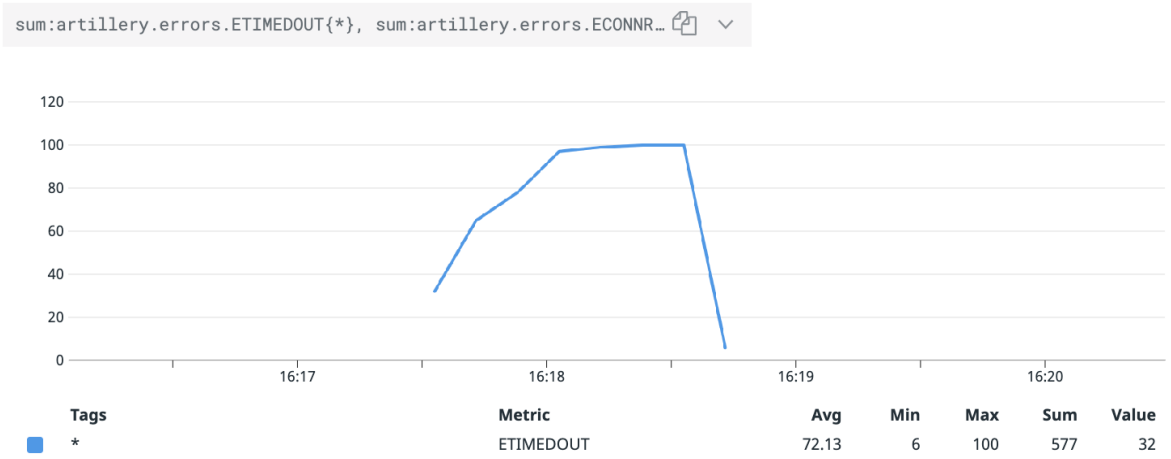


Vemos aquí que de 622 escenarios creados, solo 44 terminan OK (un 7%)



Aquí hay otro punto interesante a observar: si bien el cliente de artillery detecta 44 escenarios completados, vemos que desde node se detectan 588 response con estado 200. Además, el CPU apenas se vio esforzado.





Conclusión

Como tenemos llamadas a un servicio externo que tiene 1 solo worker y es sincrónico, solo puede procesar 1 llamada a la vez. Por lo tanto, el cuello de botella claramente es el servicio externo y el tiempo que retrasa la respuesta de l. Lo que vemos es que cada tiempo de respuesta de node va incrementando por aquellos llamados que quedan encolados, y llega un momento que los errores de ETIMEDOUT de parte del ELB empiezan a hacer fallar todos los escenarios de artillery. es llamativo que vemos alrededor de 588 responses con status 200, pero solo vemos 5 scenarios completados OK del lado de cliente.

Una hipótesis de los errores de rechazo y timeout es que la aplicación de de node se llena con request y empieza a rechazar además de que tarda mucho tiempo en responder debido a que la aplicación de python solo puede con una request a la vez.

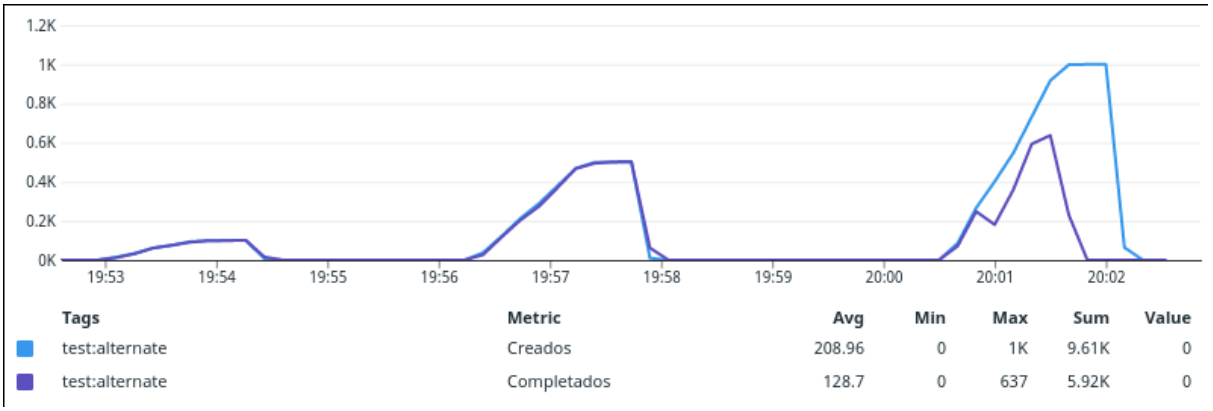
Alternate

Se usaron las siguientes escalas de RPS:

- 10
- 50
- 100

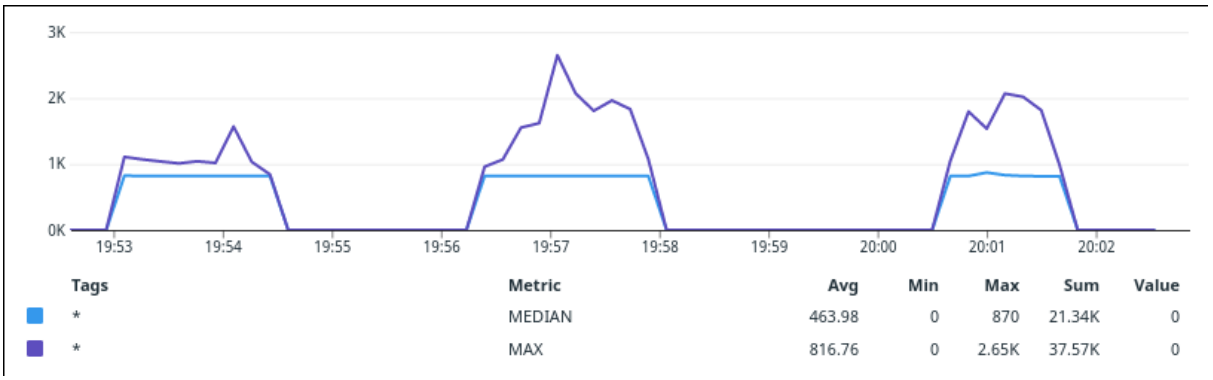
A continuación se adjuntan gráficos sobre distintas métricas a lo largo de las pruebas. El orden de ejecución se dio de menor a mayor RPS.

Escenarios de artillery creados y completados:



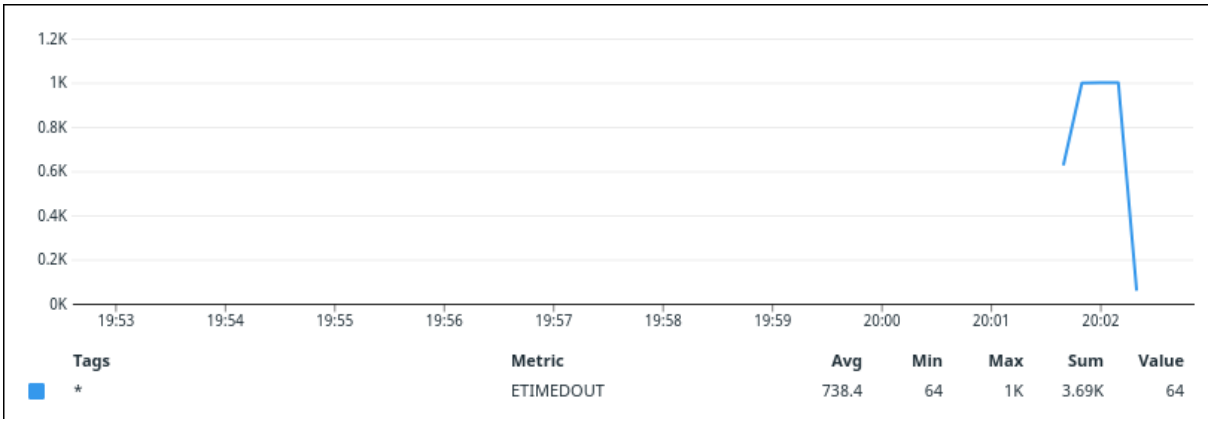
Tags	Metric	Avg	Min	Max	Sum	Value
test:alternate	Creados	208.96	0	1K	9.61K	0
test:alternate	Completados	128.7	0	637	5.92K	0

Tiempos medios y máximos de peticiones:



Tags	Metric	Avg	Min	Max	Sum	Value
*	MEDIAN	463.98	0	870	21.34K	0
*	MAX	816.76	0	2.65K	37.57K	0

Errores experimentados durante los escenarios:



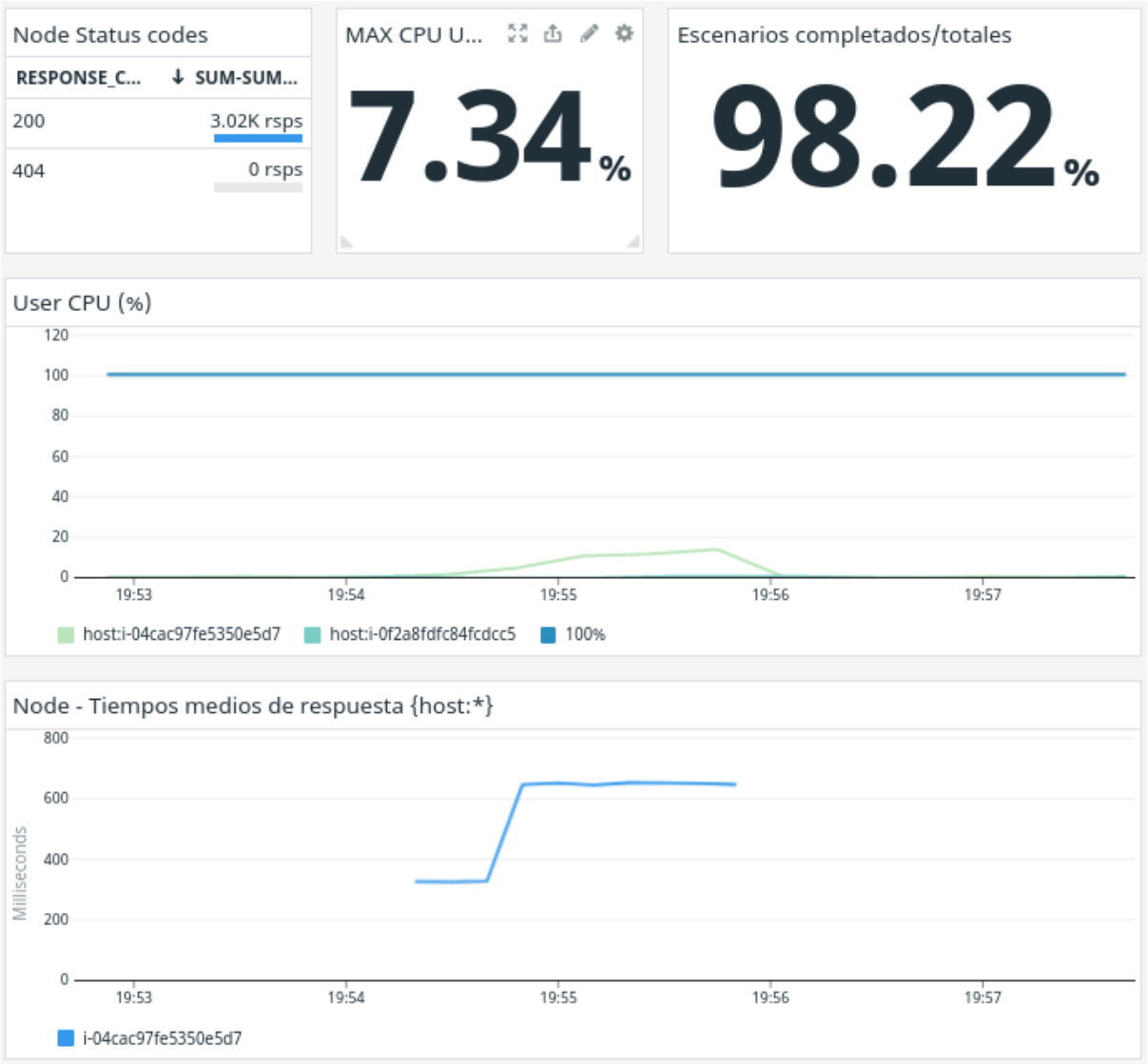
Métricas para cada una de las pruebas respecto a:

- Códigos de estado de las peticiones creadas por artillery
- Uso máximo de CPU
- Porcentaje de escenarios completados (totales de las 3 pruebas)
- Uso de CPU a lo largo del tiempo
- Tiempos medios de respuesta registrados por el router de la librería Express

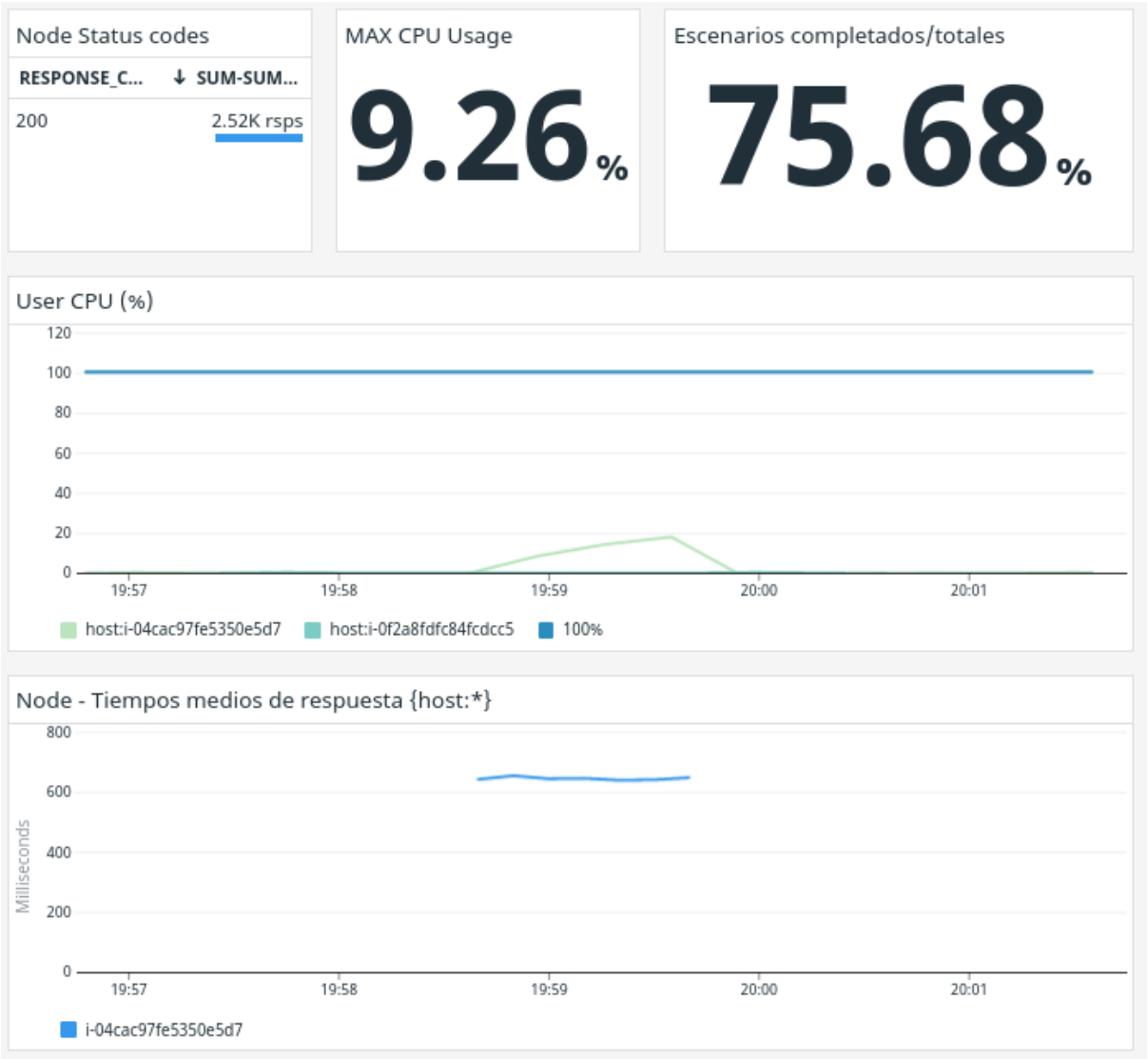
10 RPS


Node Status codes
RESPONSE_C...
SUM-SUM...
200
707 rsps
404
1 rsp
MAX CPU Usage
2.22%
Escenarios completados/totales
100.00%
User CPU (%)
120
100
80
60
40
20
0
19:50
19:51
19:52
19:53
19:54
host:i-04cac97fe5350e5d7
host:i-0f2a8fdfc84fcdcc5
100%
Node - Tiempos medios de respuesta {host:*}
500
400
300
200
100
0
Milliseconds
19:50
19:51
19:52
19:53
19:54
i-04cac97fe5350e5d7

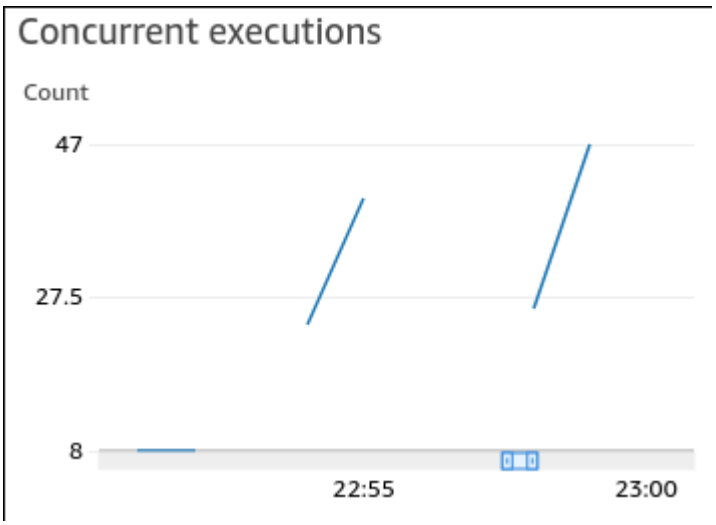
50 RPS


Node Status codes
RESPONSE_C...
SUM-SUM...
200
3.02K rsps
404
0 rsps
MAX CPU U...
7.34%
Escenarios completados/totales
98.22%
User CPU (%)
120
100
80
60
40
20
0
19:53
19:54
19:55
19:56
19:57
host:i-04cac97fe5350e5d7
host:i-0f2a8fdfc84fcdcc5
100%
Node - Tiempos medios de respuesta {host:*}
800
600
400
200
0
Milliseconds
19:53
19:54
19:55
19:56
19:57
i-04cac97fe5350e5d7

100 RPS



Ejecuciones concurrentes de la función lambda durante las pruebas



Conclusión

Se puede observar que la estrategia de utilizar el servicio lambda es más rápida que el server de python (invocado cuando se va al endpoint remote) ya que este solo tiene un worker, por cada invocación a lambda tendríamos “un worker” y eso hace que no tengamos el cuello de botella en el servicio de python. En este caso se pudo soportar un poco más de tráfico, aunque se siguen viendo errores posiblemente por las mismas razones que en el primer punto.

Configuración con cache

Se usa la configuración con 1 instancia. Con cada configuración de RPS, se concluye limpiando el caché.

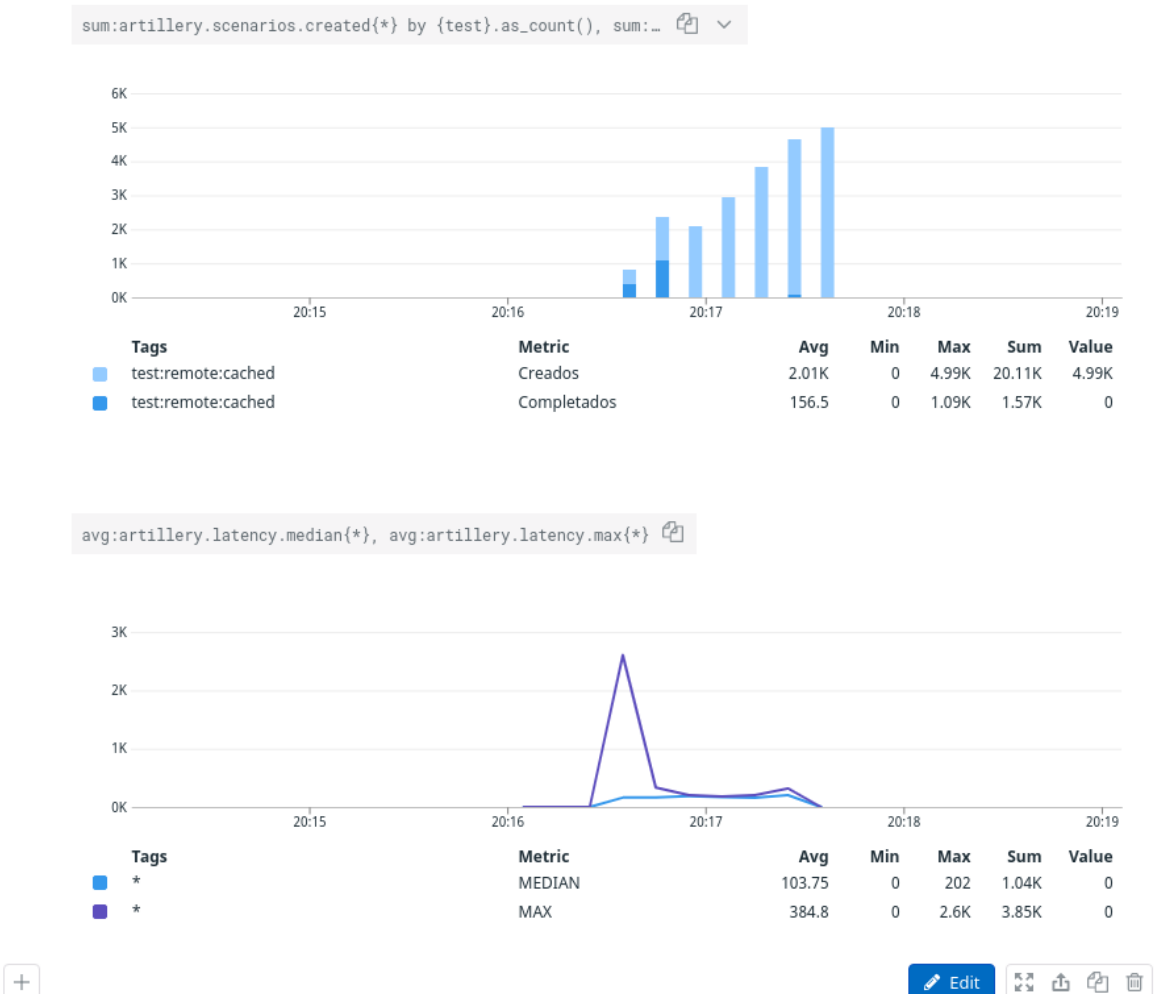
Para la mayoría de casos se usaron las siguientes escalas de RPS:

- 10
- 50
- 100
- 500

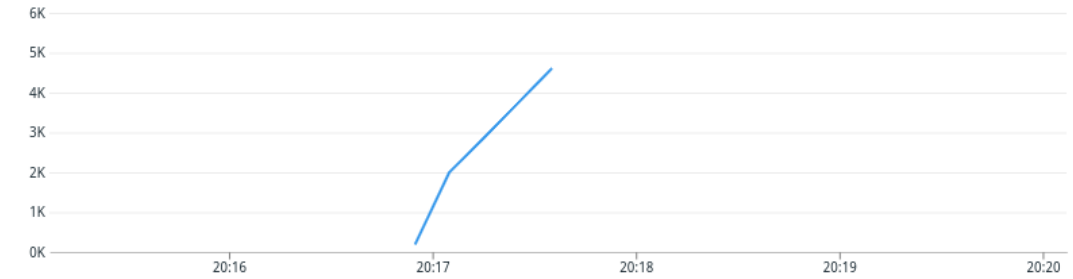
Caché con 1 sola key

Aquí obtuvimos solo una configuración con 500RPS dado que queríamos validar específicamente si existía alguna dependencia con la cantidad de keys que soportaba el caché.

Solo 500 RPS:



sum:artillery.errors.ETIMEDOUT{*}, sum:artillery.errors.ECONN…



Tags	Metric	Avg	Min	Max	Sum	Value
*	ETIMEDOUT	2.67K	216	4.59K	13.37K	4.59K

Node Status codes

RESPONSE_CODE	SUM-SUM:NODE.E...
200	1.57K rsps
204	1 rsp
500	0 rsps

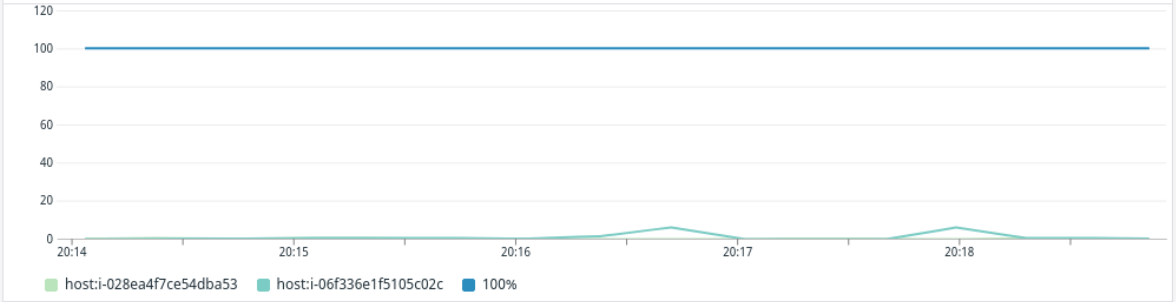
MAX CPU Usage

3.18%

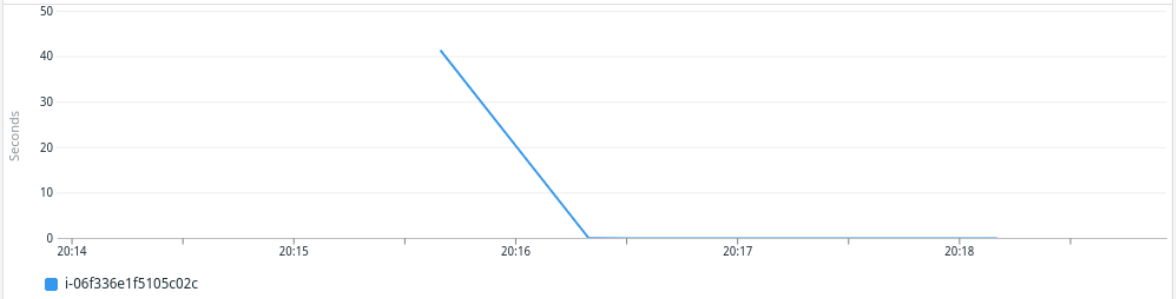
Escenarios completados/totales

7.78%

System CPU (%)

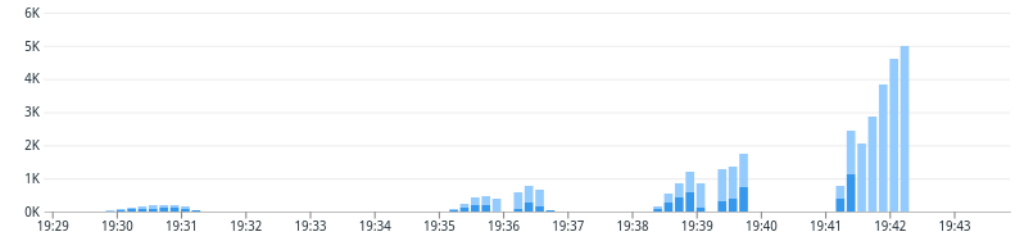


Node - Tiempos medios de respuesta {host:*}



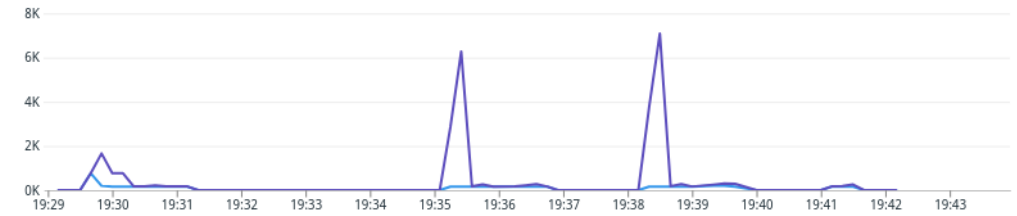
Caché con 10 keys

sum:artillery.scenarios.created{*} by {test}.as_count(), sum:…



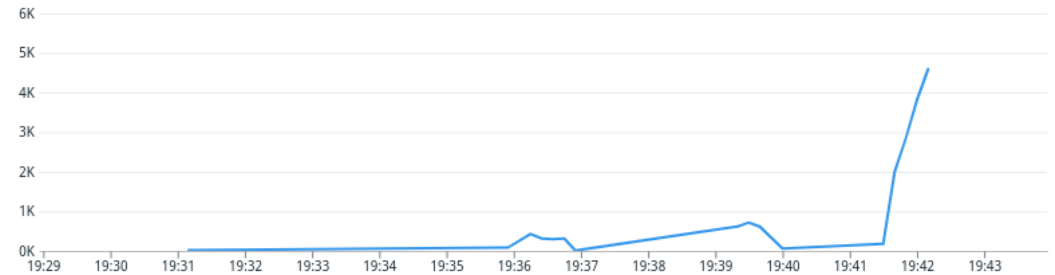
Tags	Metric	Avg	Min	Max	Sum	Value
test:remote:cached	Creados	542.77	0	5K	28.22K	92
test:remote:cached	Completados	116.31	0	1.13K	6.05K	91

avg:artillery.latency.median{*}, avg:artillery.latency.max{*}



Tags	Metric	Avg	Min	Max	Sum	Value
*	MEDIAN	106.69	0	765	5.55K	0
*	MAX	549.23	0	7.09K	28.56K	0

sum:artillery.errors.ETIMEDOUT{*}, sum:artillery.errors.ECONN…



Tags	Metric	Avg	Min	Max	Sum	Value
*	ETIMEDOUT	1.06K	10	4.59K	16.89K	4.59K

Node Status codes

RESPONSE_CODE	SUM-SUM:NODE.E...
200	6.19K rsps
404	7 rsps
204	4 rsps

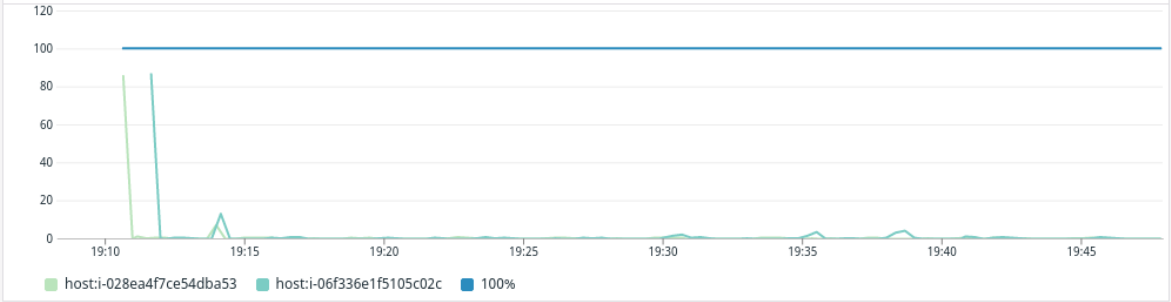
MAX CPU Usage

85.33%

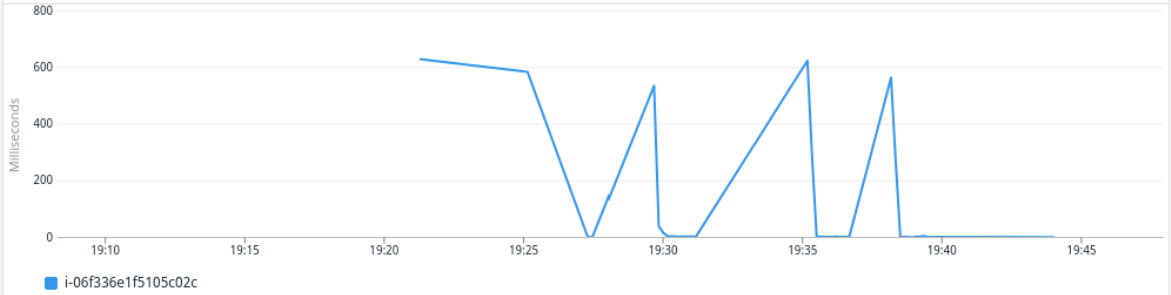
Escenarios completados/totales

18.17%

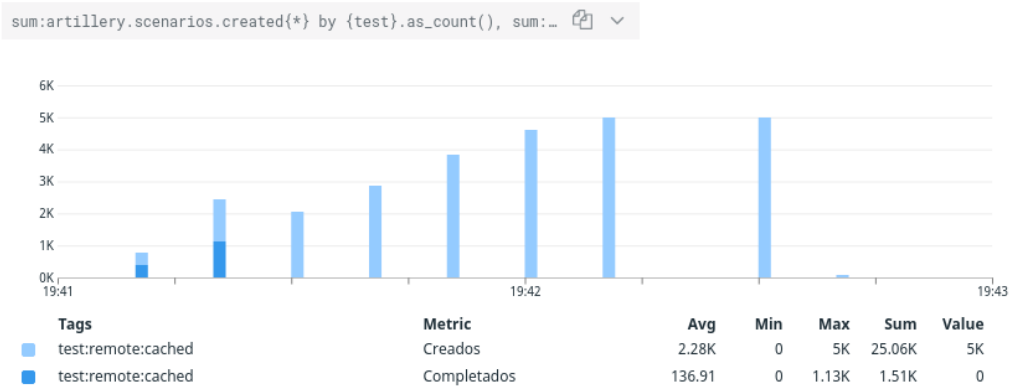
System CPU (%)

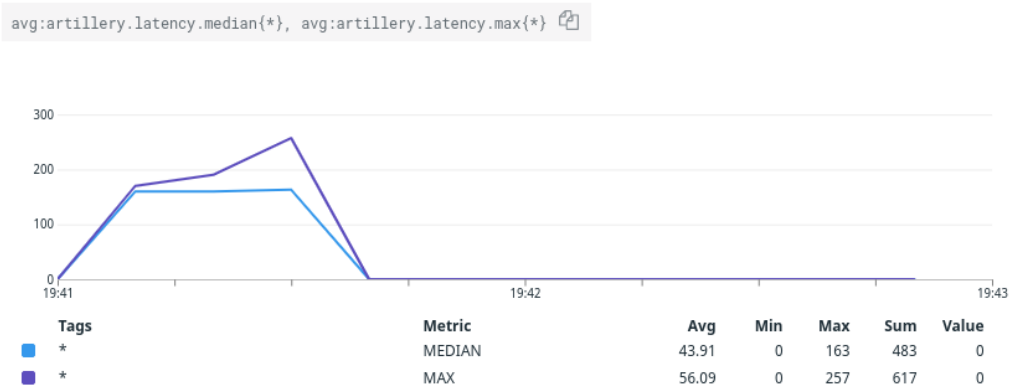


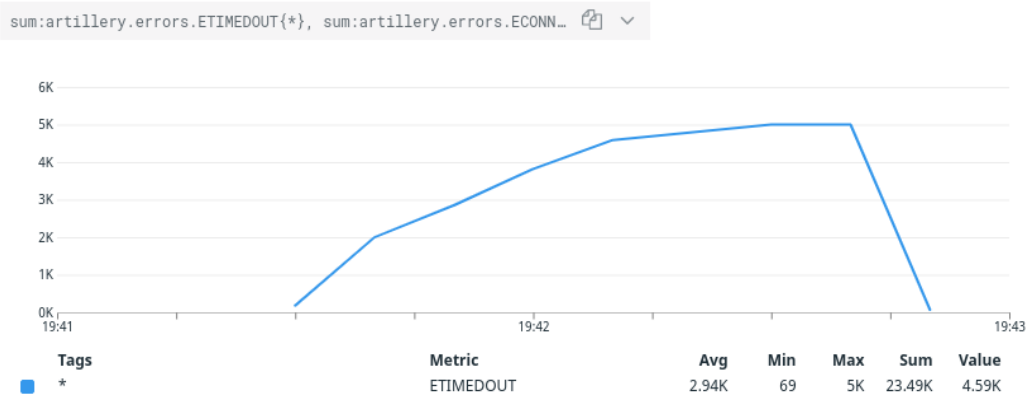
Node - Tiempos medios de respuesta {host:*}



Solo 500 rps:


sum:artillery.scenarios.created{*} by {test}.as_count(), sum:…
6K
5K
4K
3K
2K
1K
0K
19:41
19:42
19:43
Tags
Metric
Avg
Min
Max
Sum
Value
test:remote:cached
Creados
2.28K
0
5K
25.06K
5K
test:remote:cached
Completados
136.91
0
1.13K
1.51K
0


avg:artillery.latency.median{*}, avg:artillery.latency.max{*}
300
200
100
0
19:41
19:42
19:43
Tags
Metric
Avg
Min
Max
Sum
Value
*
MEDIAN
43.91
0
163
483
0
*
MAX
56.09
0
257
617
0


sum:artillery.errors.ETIMEDOUT{*}, sum:artillery.errors.ECONN…
6K
5K
4K
3K
2K
1K
0K
19:41
19:42
19:43
Tags
Metric
Avg
Min
Max
Sum
Value
*
ETIMEDOUT
2.94K
69
5K
23.49K
4.59K

Node Status codes

RESPONSE_CODE	SUM-SUM:NODE.E...
200	1.51K rsps
404	0 rsps
204	0 rsps

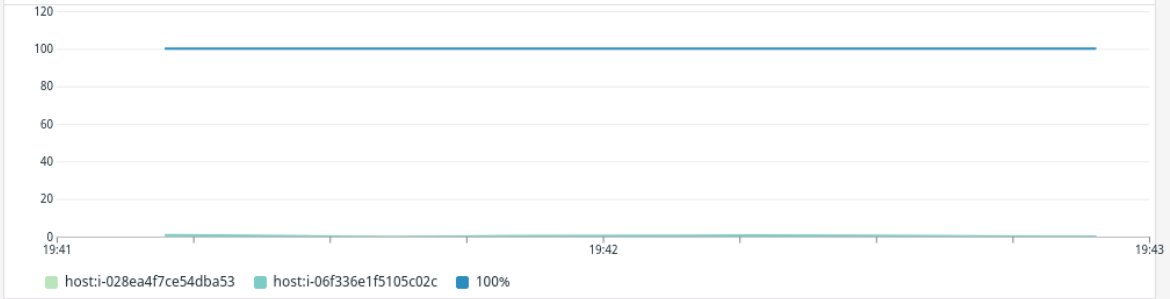
MAX CPU Usage

0.89%

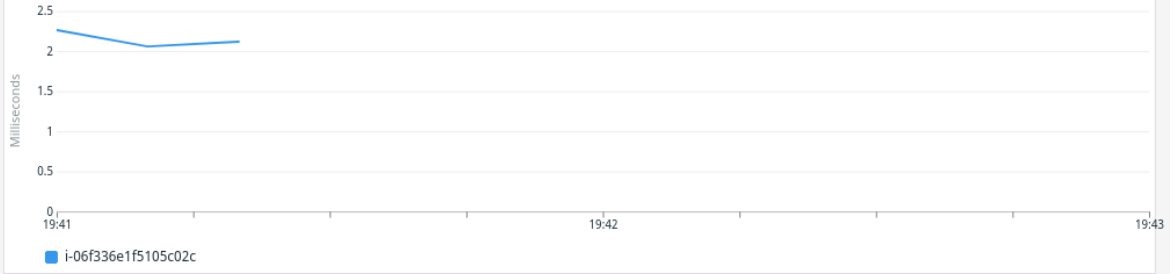
Escenarios completados/totales

6.01%

System CPU (%)

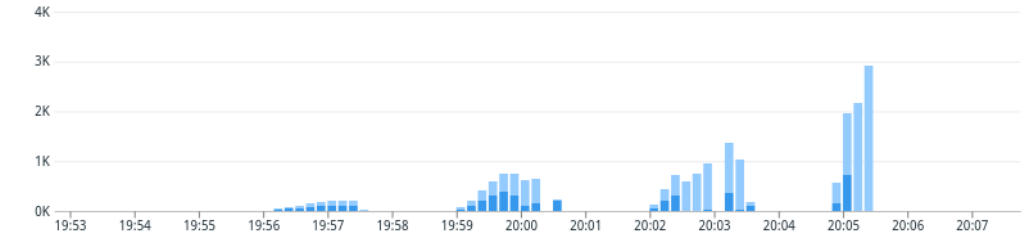


Node - Tiempos medios de respuesta {host:*}



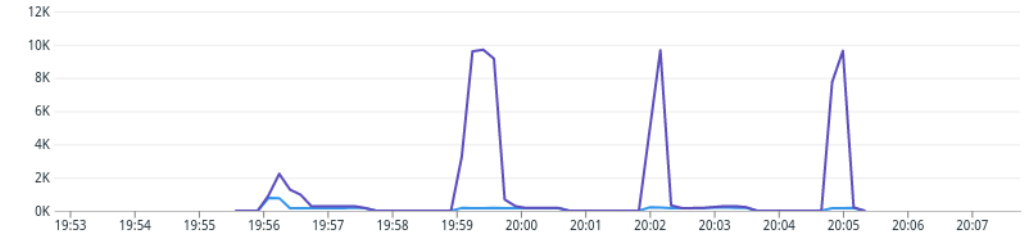
Caché con 20 keys

sum:artillery.scenarios.created{*} by {test}.as_count(), sum:…



Tags	Metric	Avg	Min	Max	Sum	Value
test:remote:cached	Creados	298.18	0	2.92K	14.91K	2.92K
test:remote:cached	Completados	87.32	0	719	4.37K	0

avg:artillery.latency.median{*}, avg:artillery.latency.max{*}



Tags	Metric	Avg	Min	Max	Sum	Value
*	MEDIAN	127.64	0	766	6.38K	0
*	MAX	1.47K	0	9.69K	73.73K	0

sum:artillery.errors.ETIMEDOUT{*}, sum:artillery.errors.ECONN…

Tags	Metric	Avg	Min	Max	Sum	Value
*	ETIMEDOUT	497.4	16	2.14K	7.46K	2.14K

Node Status codes

RESPONSE_CODE	↓ SUM-SUM:NODE.E...
200	5.17K rsps
204	4 rsps
404	1 rsp

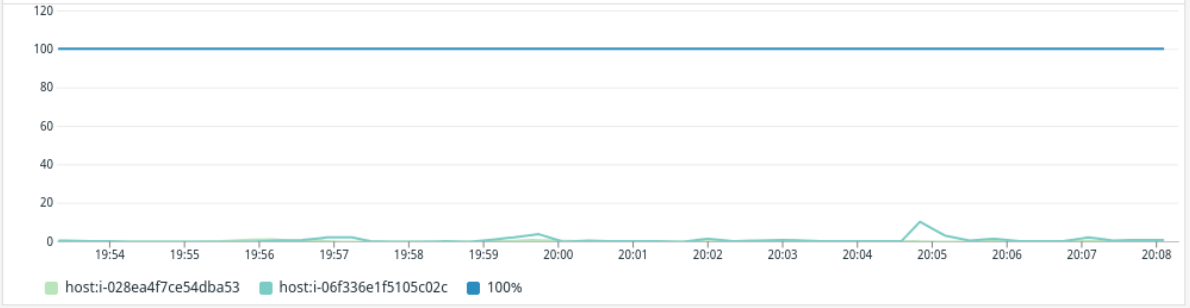
MAX CPU Usage

5.38%

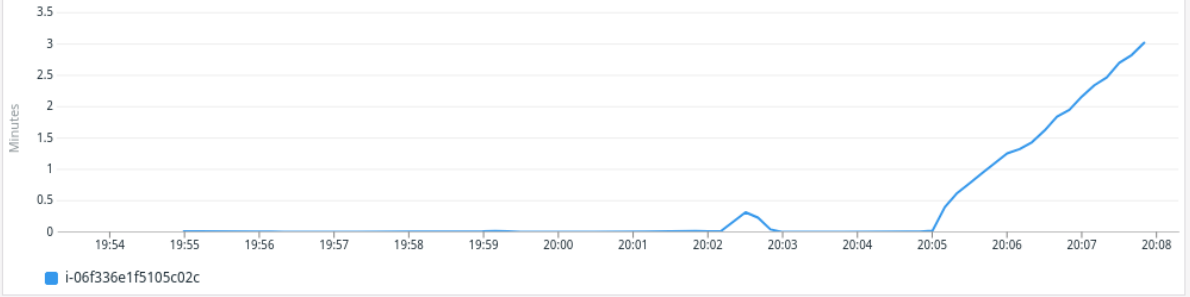
Escenarios completados/totales

29.28%

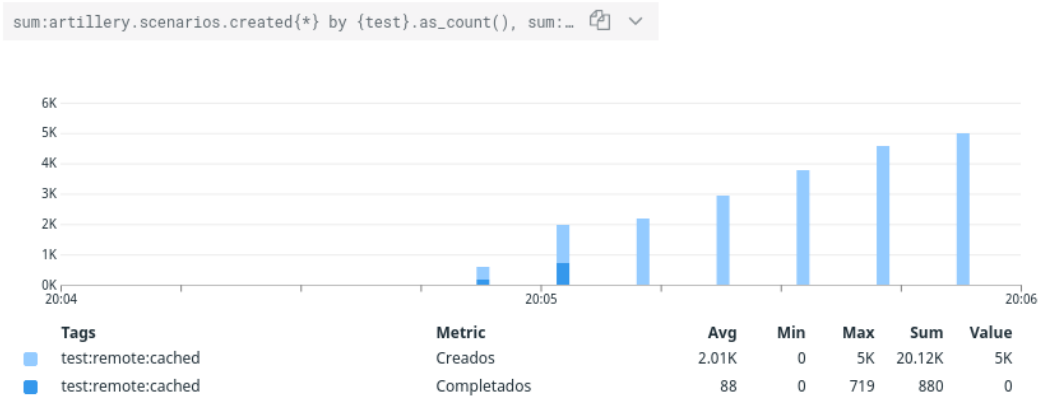
System CPU (%)



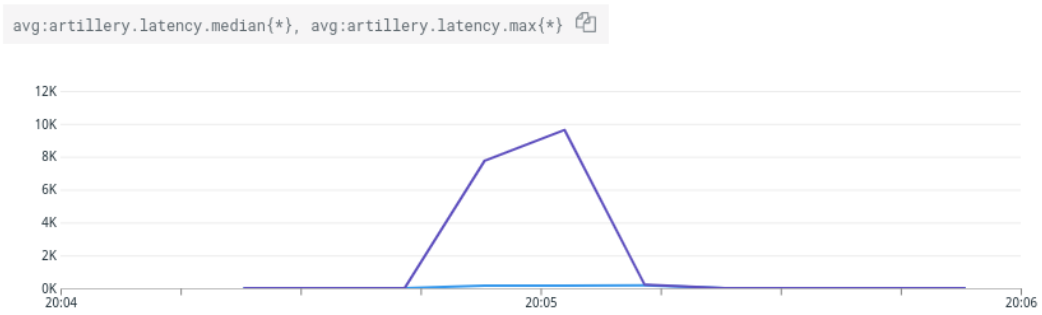
Node - Tiempos medios de respuesta {host:*}



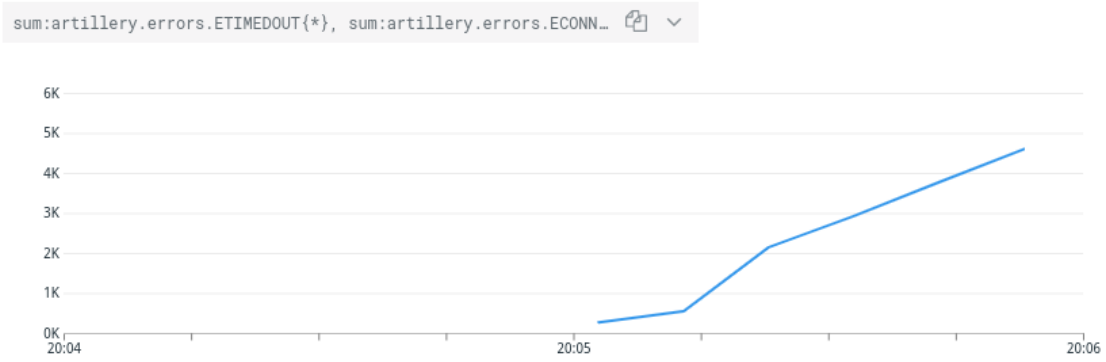
Solo 500 RPS:



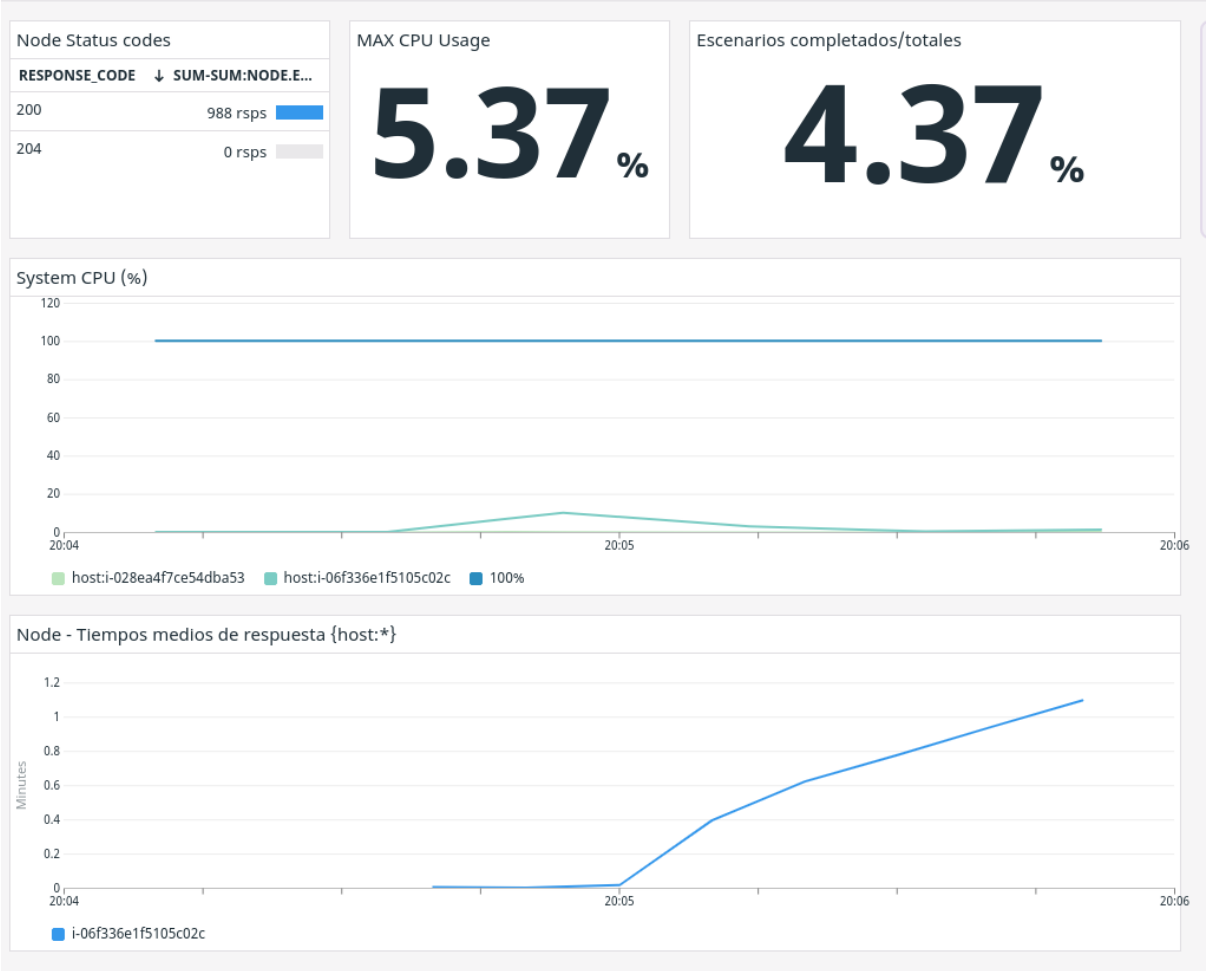
Tags	Metric	Avg	Min	Max	Sum	Value
test:remote:cached	Creados	2.01K	0	5K	20.12K	5K
test:remote:cached	Completados	88	0	719	880	0



Tags	Metric	Avg	Min	Max	Sum	Value
*	MEDIAN	48.8	0	165	488	0
*	MAX	1.76K	0	9.63K	17.58K	0



Tags	Metric	Avg	Min	Max	Sum	Value
*	ETIMEDOUT	2.37K	263	4.59K	14.22K	3.76K



Conclusión:

Se puede observar que a mayor tamaño del cache peor es la performance a mayor RPS, esto puede ser debido a que la aplicación no puede manejar bien el indexado de las entradas o el cache cae en performance con el mayor tamaño.

También podemos ver que la performance mejora con el caché, ya que se completan más request hasta aproximadamente 100RPS, aliviana un poco pero igual tiene su límite.

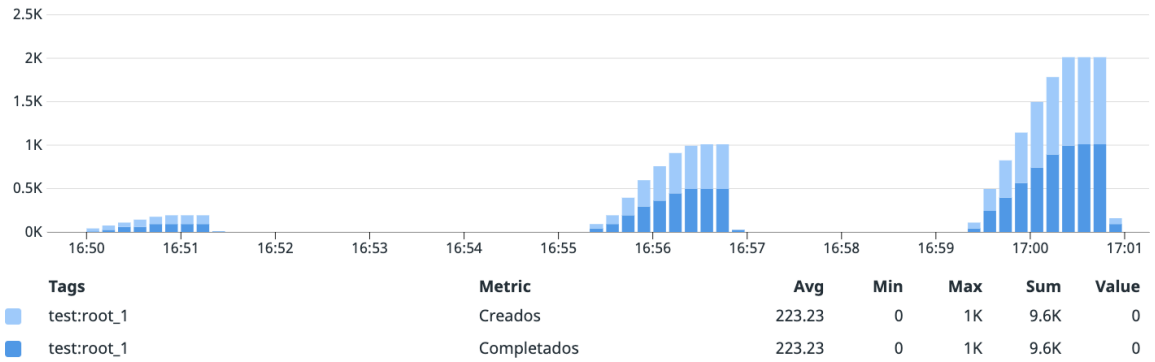
Configuración con 3 instancias

Root

Configuración de artillery: Se corren los mismos escenarios que para 1 instancia

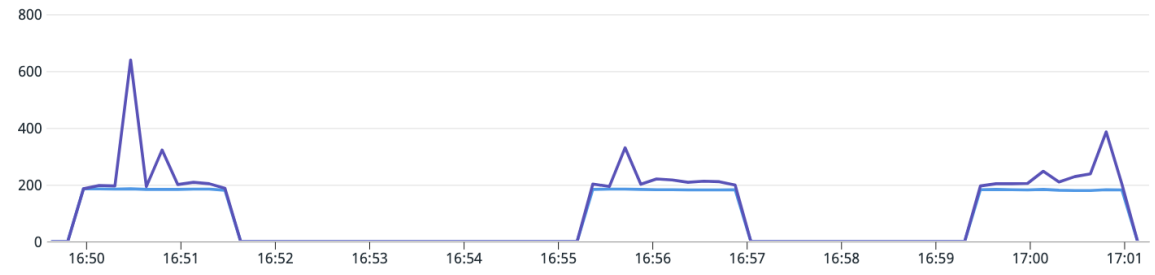
10, 50 y 100 RPS

```
sum:artillery.scenarios.created{*} by {test}.as_count(), sum:a…
```



Tags	Metric	Avg	Min	Max	Sum	Value
test:root_1	Creados	223.23	0	1K	9.6K	0
test:root_1	Completados	223.23	0	1K	9.6K	0

```
sum:artillery.latency.median{*}, sum:artillery.latency.max{*}
```



Tags	Metric	Avg	Min	Max	Sum	Value
*	MEDIAN	125.01	0	186	5.5K	0
*	MAX	160.57	0	641	7.07K	0

Node Status codes

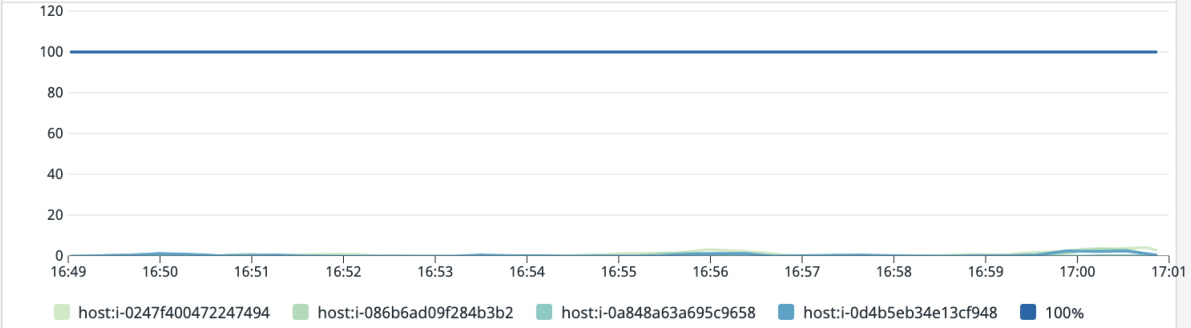
RESPONSE_CO...	↓ SUM-SUM:...
200	9.6K rsps
404	0 rsps

MAX CPU Usage

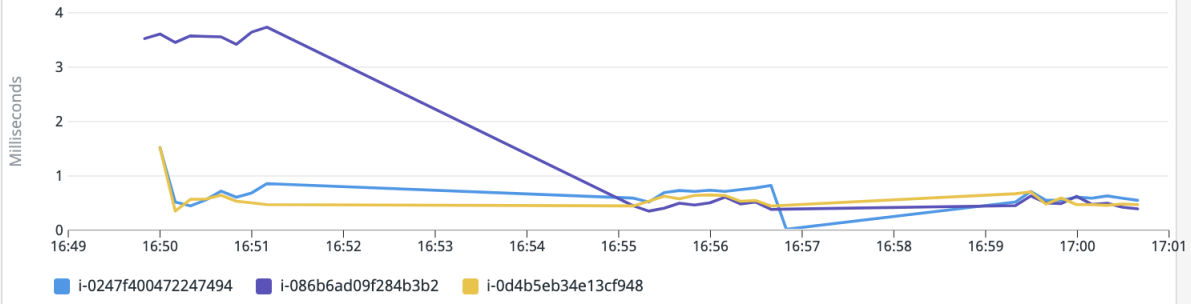
2.55%

Add widgets

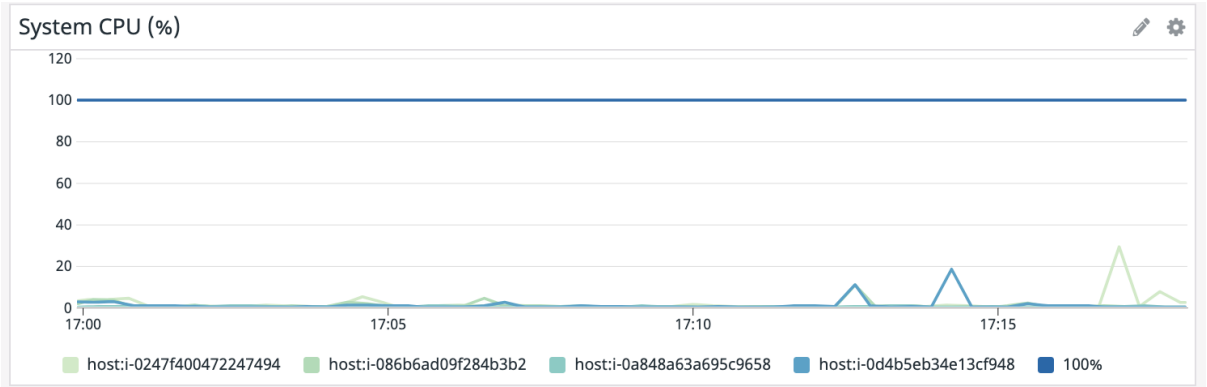
System CPU (%)

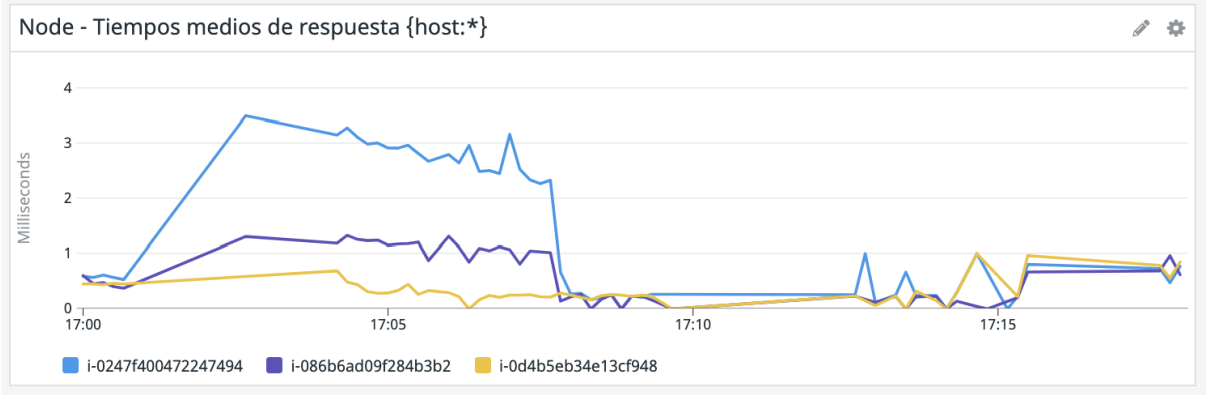


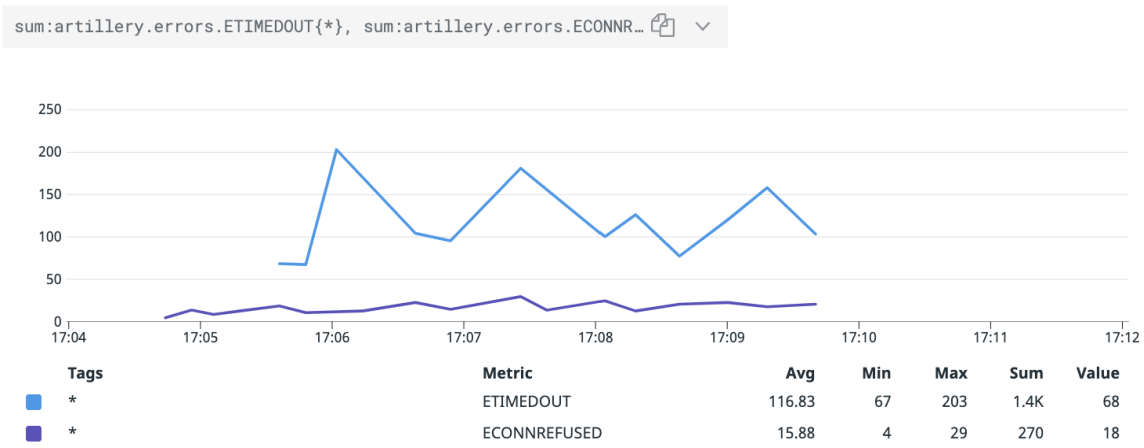
Node - Tiempos medios de respuesta {host:*}

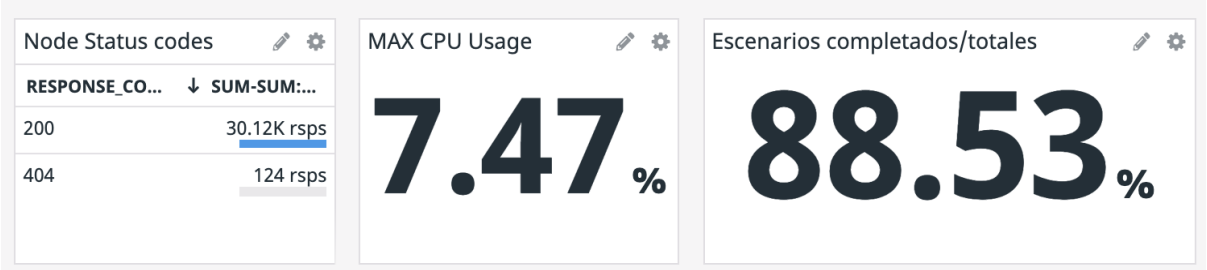


Ejemplo con rampto 400 y plain 500





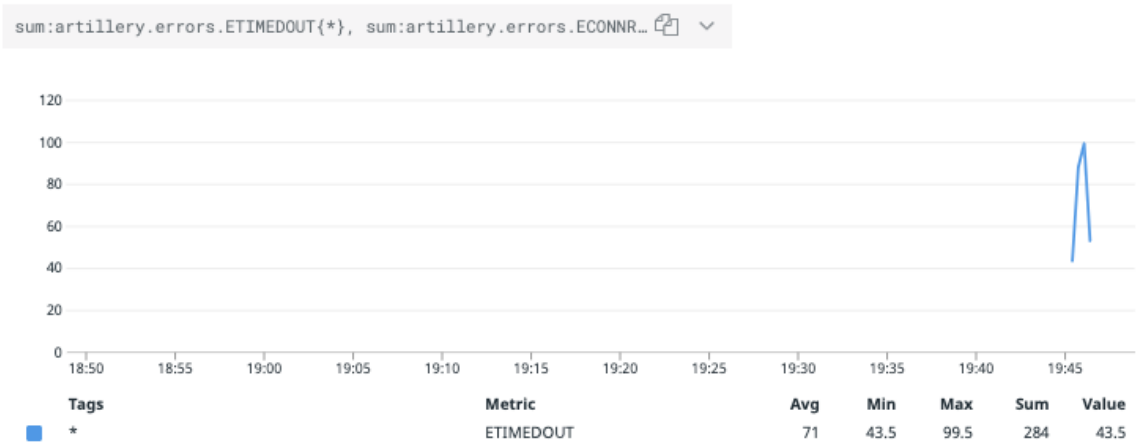


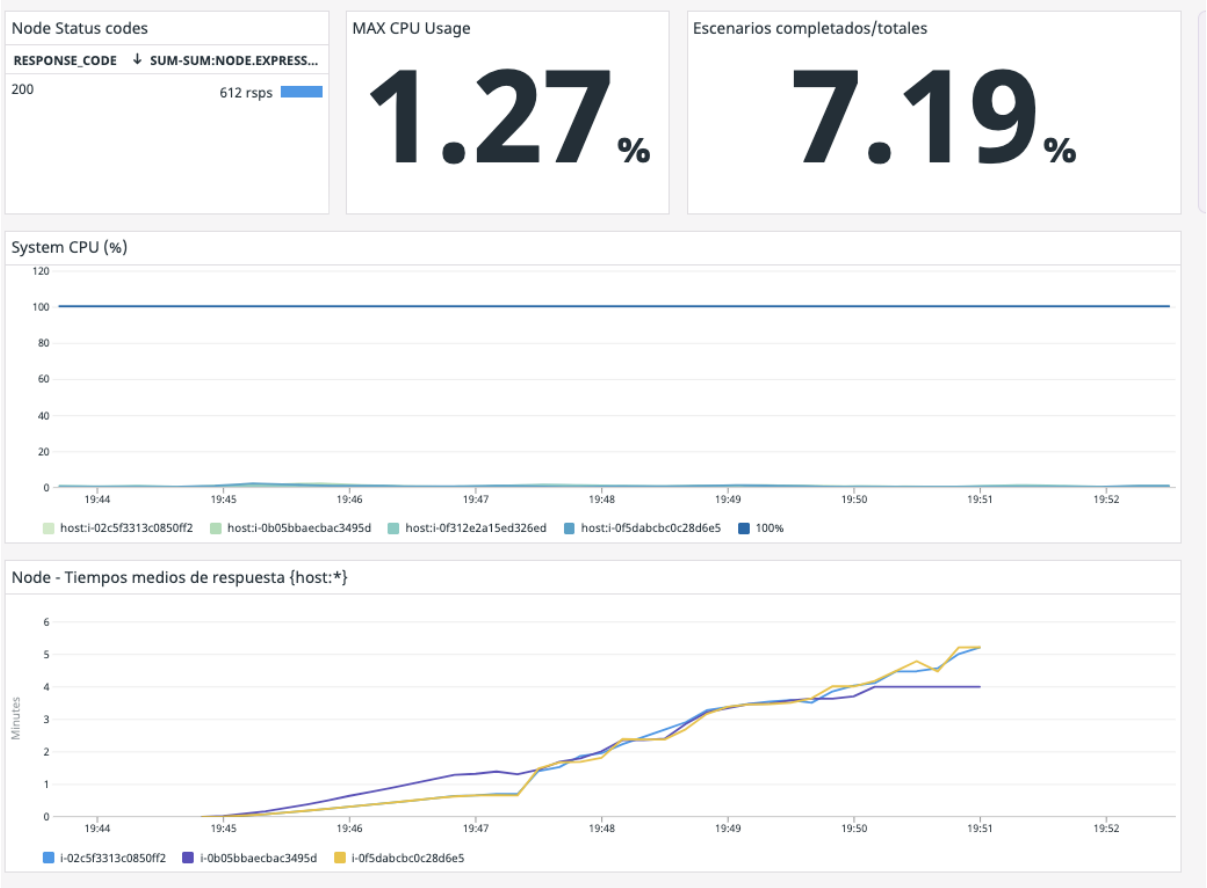


Conclusión

Se ve una mejora frente a la versión de 1 instancia, puede vislumbrarse en la diferencia entre la cantidad de errores. Además se puede ver que el uso de CPU promedio en las instancias es más bajo.

Remote





Conclusión

Vemos que los escenarios dejaron de disparar alrededor de las 19:46, y en node los tiempos de respuestas siguen altos hasta las 19.51 y más. Eso es porque quedaron esos request pendientes debido a que hay un solo server de python operando sincrónicamente con 1 solo worker.

Escalar en el servidor no cambia la performance ya que el cuello de botella es la aplicación de python.

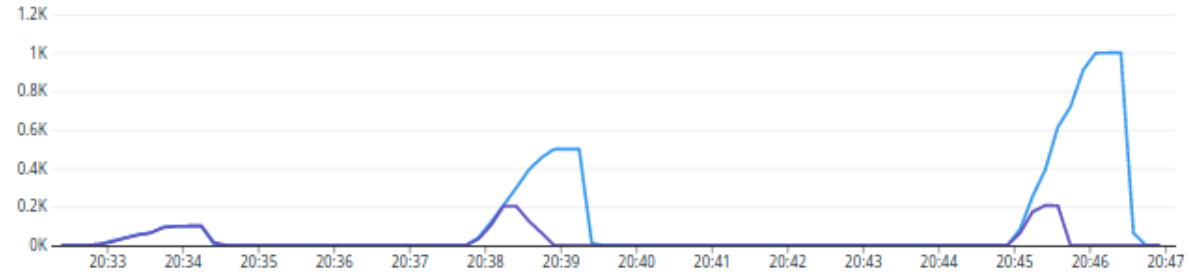
Alternate

Se usaron las siguientes escalas de RPS:

- 10
- 50
- 100

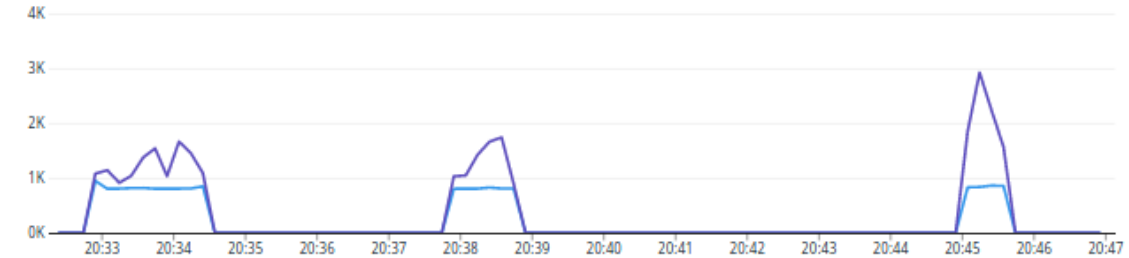
A continuación se adjuntan gráficos sobre distintas métricas a lo largo de las pruebas. El orden de ejecución se dio de menor a mayor RPS.

Escenarios creados y completados:



Tags	Metric	Avg	Min	Max	Sum	Value
test:alternate	Creados	214.31	0	1K	9.64K	0
test:alternate	Completados	43.89	0	207	1.98K	0

Tiempos medios y máximos de peticiones:



Tags	Metric	Avg	Min	Max	Sum	Value
*	MEDIAN	365.73	0	946.5	16.46K	0
*	MAX	636.91	0	2.92K	28.66K	0

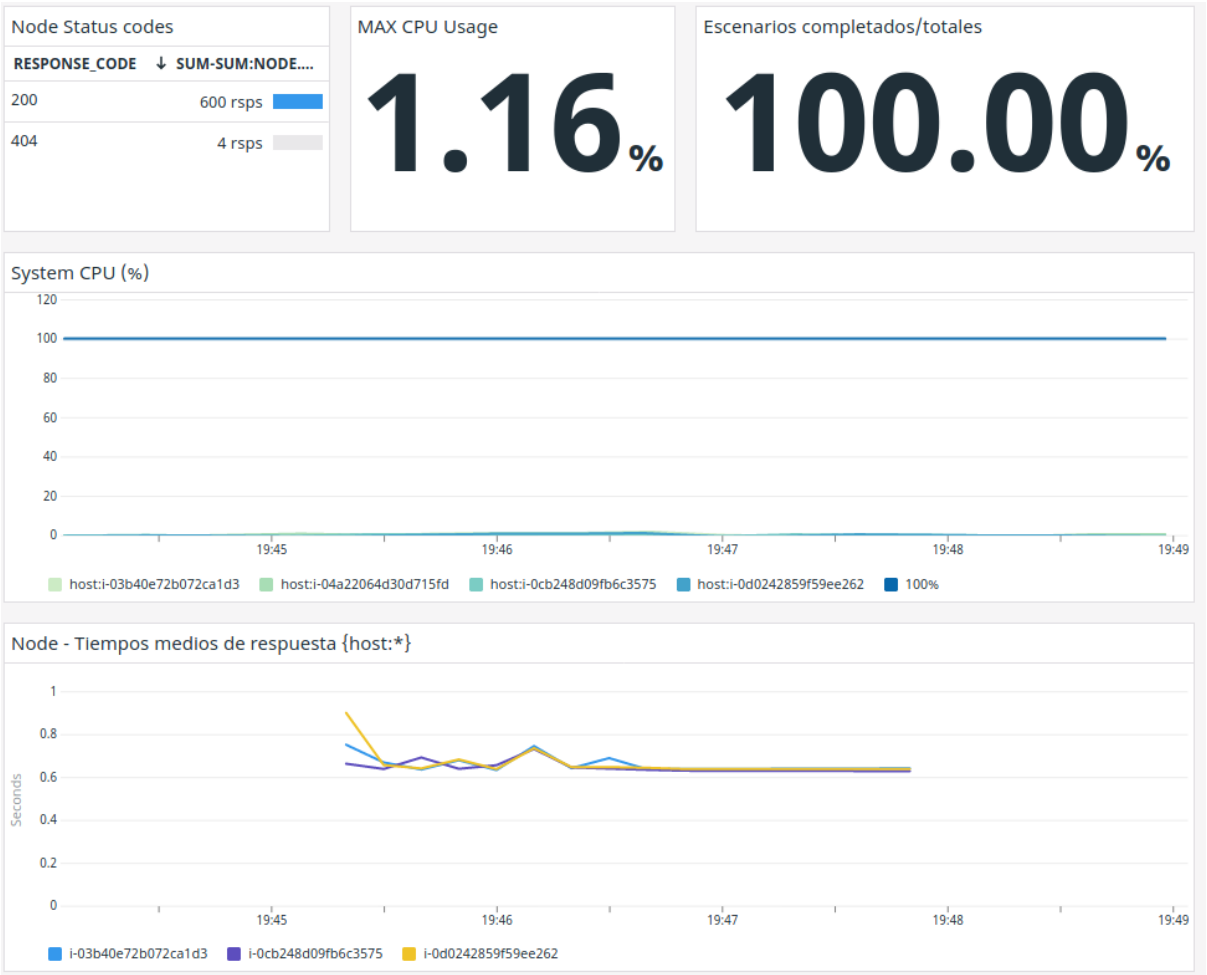
Errores experimentados durante los escenarios:

Tags	Metric	Avg	Min	Max	Sum	Value
*	ETIMEDOUT	547.79	10	1K	7.67K	322

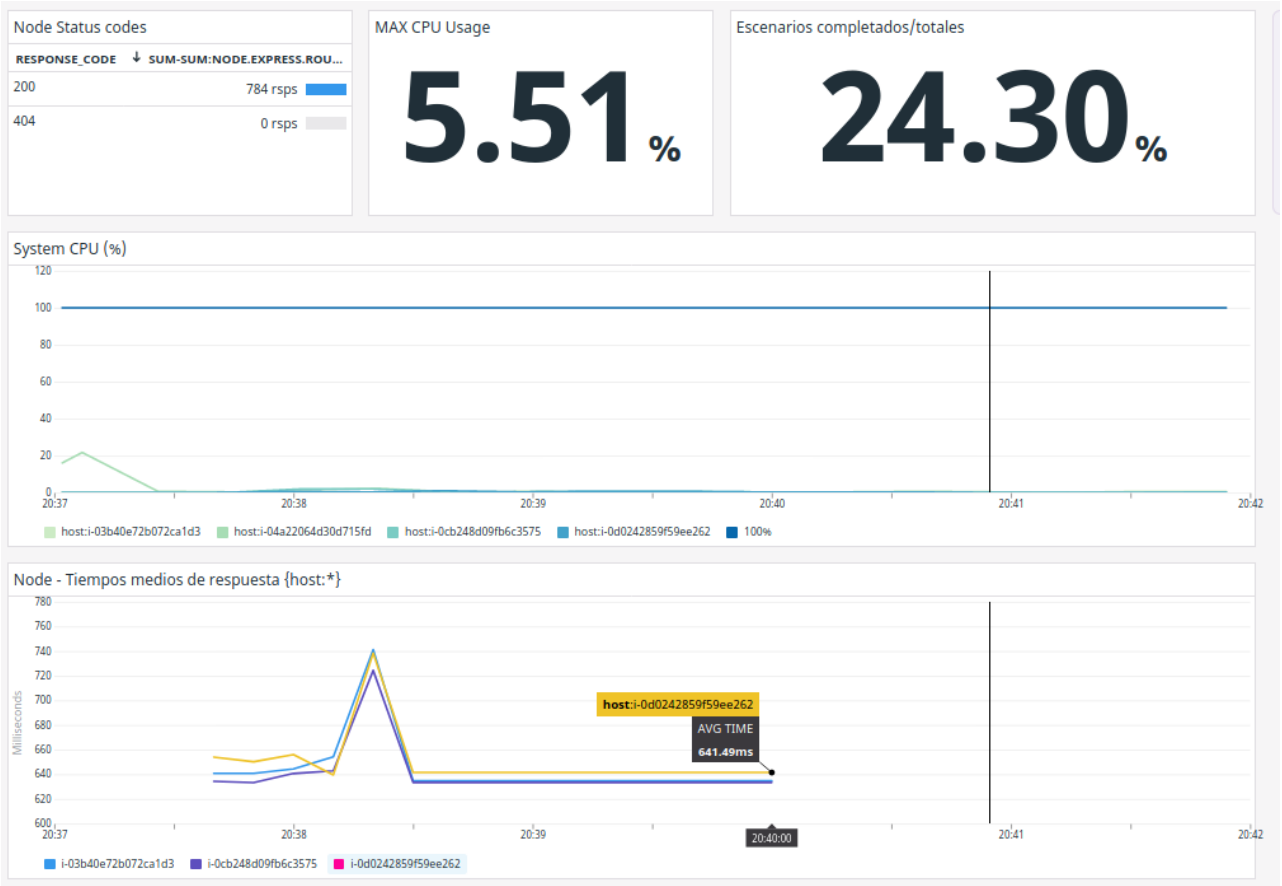
Métricas para cada una de las pruebas respecto a:

- Códigos de estado de las peticiones creadas por artillery
- Uso máximo de CPU
- Porcentaje de escenarios completados (totales de las 3 pruebas)
- Uso de CPU a lo largo del tiempo
- Tiempos medios de respuesta registrados por el router de la librería Express

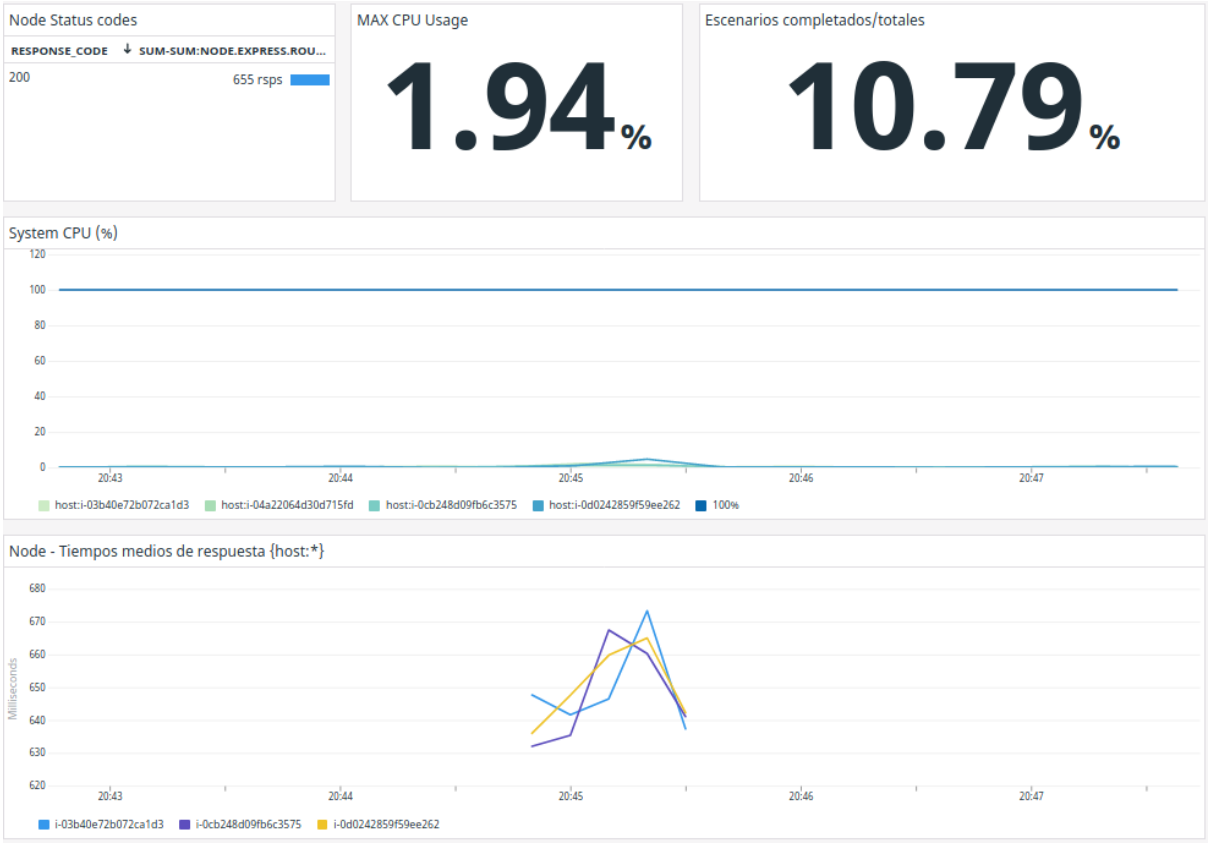
10 RPS



50 RPS


Node Status codes
RESPONSE_CODE
SUM-SUM:NODE.EXPRESS.ROU...
200
784 rsps
404
0 rsps
MAX CPU Usage
5.51%
Escenarios completados/totales
24.30%
System CPU (%)
120
100
80
60
40
20
0
20:37
20:38
20:39
20:40
20:41
20:42
host:i-03b40e72b072ca1d3
host:i-04a22064d30d715fd
host:i-0cb248d09fb6c3575
host:i-0d0242859f59ee262
100%
Node - Tiempos medios de respuesta {host:*}
Milliseconds
780
760
740
720
700
680
660
640
620
600
host:i-0d0242859f59ee262
AVG TIME
641.49ms
20:40:00
i-03b40e72b072ca1d3
i-0cb248d09fb6c3575
i-0d0242859f59ee262

100 RPS



Ejecuciones concurrentes de la función lambda durante las pruebas

Conclusión

Se puede ver que al escalar horizontalmente se tiene una mayor capacidad. Mirando la corrida con 100 RPS vemos una concurrencia de lambdas de 78 (mientras en la versión de alternate de 1 sola instancia se llega a 47 ejecuciones concurrentes). Esto es debido a que

en el caso anterior el cuello de botella era la función de python, y en este caso se volvió la app de node el cuello de botella.

Además se observa que tanto para el segundo escenario como para el tercero la cantidad de escenarios completados es aproximadamente el mismo, por lo que ese sería el límite.

Resultados finales

Después de realizadas las pruebas podemos, vemos el porcentaje de escenarios completados y tratamos de estimar los límites de los endpoints:

Configuración	Root	Remote	Alternate
1 Instancia	Hasta 300 RPS	Menos de 10 RPS	Hasta 45 RPS
1 Instancia con cache	---N/A---	Hasta 100 RPS	---N/A---
3 Instancias	Hasta 500 RPS	Menos de 10 RPS	Hasta 80 RPS

En general se puede ver que la adopción de la estrategia de lambda mejora ampliamente la eficiencia y esta puede ser mejorada escalando horizontalmente el servidor web de node.

Vimos en varias partes errores que no son concluyentes el origen, ya que pueden ser de varios orígenes como:

- Artillery corriendo en el local y la pc personal siendo un cuello de botella
- La conexión de internet en donde se corrieron las pruebas
- La capacidad de las máquinas micro (en manejo de red y conexiones)