

# Trabajo Práctico 2

Facultad de Ingeniería, Universidad de Buenos Aires  
[75.73] Arquitectura de Software

Grupo “Ladrillo”

100029 del Mazo, Federico  
102264 Hojman, Joaquin  
97112 Kasman, Lucía  
97131 Rombolá, Juan Pablo  
103409 Schmidt, Agustina

23 de Noviembre de 2022

## Trabajo Práctico 2

En este trabajo buscamos hacer un análisis de un sistema en distintas configuraciones, intentando encontrar el cuello de botella que lo limita.

Específicamente, nuestro sistema se compone de un cluster de instancias de aplicaciones de **express** (**node.js**) que a su vez llaman a un servidor de **gunicorn** (**python**).

```
# Pingueamos a nuestro Azure Virtual Machine Scale Set
# Este cluster funciona como load balancer y delega el ping a alguna de las instancias
↪ de node.js
$ curl "http://ladrillo-fdm.eastus.cloudapp.azure.com"
Hello World!

# Pingueamos al VMSS, pero esta vez le pedimos a `/remote` en vez de `/`
# Este endpoint hace un llamado externo al servidor de gunicorn
$ curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote"
{"id":1}
```

Las tres configuraciones que queremos analizar son:

- ¿Qué pasa si sólo tenemos una instancia de **node**?
- ¿Qué pasa si a una única instancia de **node** le agregamos una cache de **Redis**?
- ¿Qué pasa si el cluster delega a 2 instancias de **node** balanceando la carga?

Nuestras pruebas consisten en correr el mismo escenario de **artillery** para todas las configuraciones de nuestro sistema, para así tener puntos de comparación.

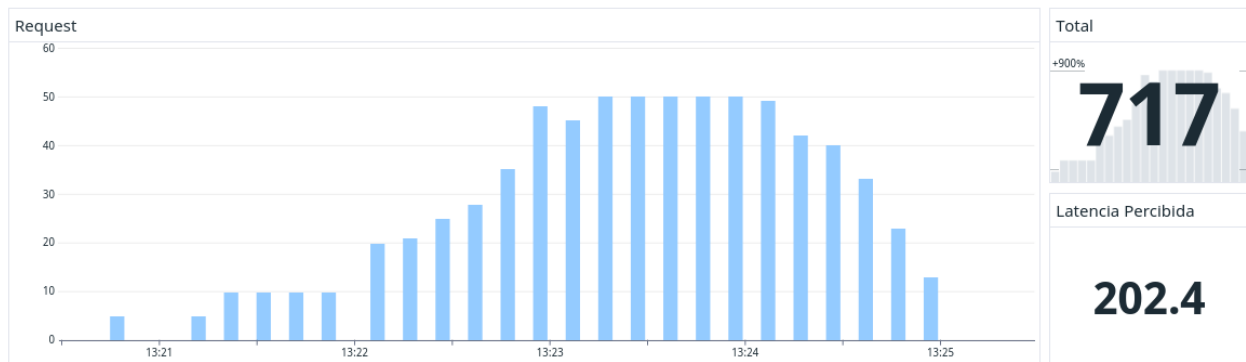
```
# Llamamos al escenario de artillery sobre el endpoint `/remote`
# El escenario consiste en correr `ping.yaml` el cual es nada más unos llamados a `/`
→ para ver cuánta latencia estamos manejando actualmente, y luego correr
→ `scenario.yaml` que contiene el flujo principal de WarmUp + RampUp + Plain +
→ Cleanup
$ cd perf
$ ./run.sh "/remote"
All VUs finished. Total time: 4 minutes, 16 seconds

-----
Summary report
-----

vusers.created: ..... 717
http.requests: ..... 717
http.request_rate: ..... 2/sec
```

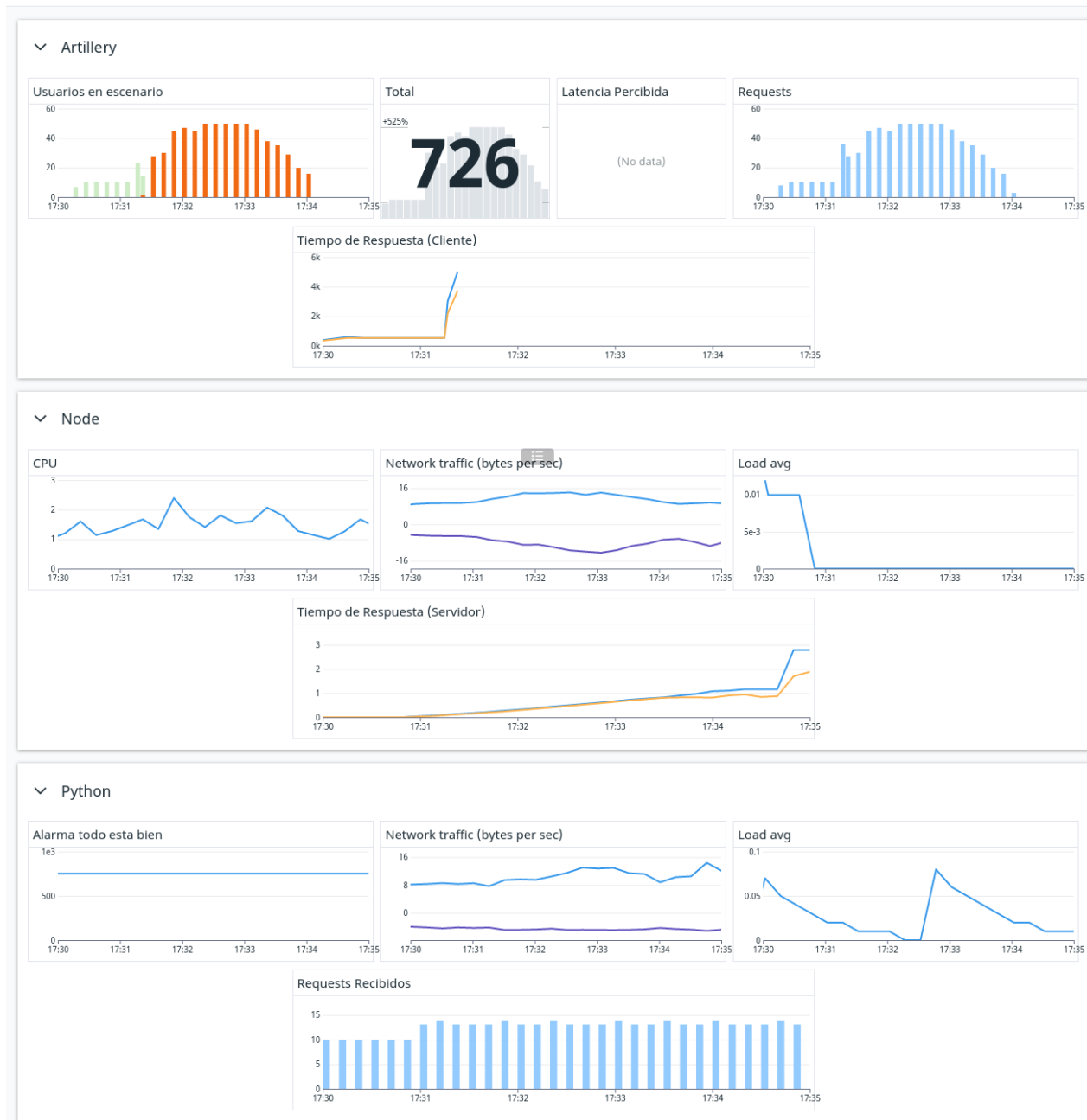
El escenario corrido finalmente se refleja en Datadog, ya que todos los componentes de nuestro sistema tienen distintos agentes que reportan métricas. A partir de ello, nos daremos una idea de dónde están los distintos cuellos de botella que limitan nuestro sistema.

▼ Artillery



Escenario de artillery a correr sobre todas las configuraciones: **WarmUp** + **RampUp** + **Plain** + **CleanUp**

Para el análisis hicimos un *dashboard*<sup>1</sup>, para ver cómo funciona y se relaciona cada componente.



Dashboard global del sistema

- Localmente, nos interesan las métricas que envía **artillery**:
  - Los usuarios completados y fallidos nos muestran dónde está el punto de quiebre del sistema.
  - Los requests por segundo nos muestran el patrón que armó nuestro escenario.
  - El tiempo de respuesta nos muestra el punto de vista del cliente, que nos sirve para compararlo con el resto del sistema.
  - El número total de requests nos ayuda a confirmar que estamos efectivamente analizando un escenario entero (en vez de uno parcialmente, o el fin de uno y el comienzo de otro).

<sup>1</sup>El dashboard fue exportado como **JSON** a `./ddog-dashboard.json` para ayudar con la réplica del estudio.

- La latencia percibida nos da una gran idea de cuánto estamos perdiendo en el trayecto desde la computadora local hasta la instancia de la VMSS, porque surge de llamados a `/` y no a `/remote`. Es decir, al restarle este número al tiempo de respuesta de `/remote`, podemos aproximar cuánto está tardando la máquina de `node` en llamar a la máquina de `python`.
- Luego tenemos las métricas de una de las instancias de `node`:
  - El tráfico de red, el consumo de CPU y el *load average* nos sirven para ver cómo está trabajando la máquina, y así poder buscar dónde se producen los picos y se está saturando.
  - El tiempo de respuesta ahora podemos verlo también desde el servidor, y compararlo con el punto de vista del cliente.
- Finalmente, tenemos los gráficos de la máquina de `gunicorn`, de la cual teóricamente no tenemos información ni acceso, pero que aun así nos es funcional al análisis:
  - La “alarma todo está bien” nos muestra que el servicio está funcionando correctamente. Viendo el código sabemos que todos los pedidos tienen un `sleep(0.75)`, es por eso que este gráfico *siempre* debe ser una línea de 750 milisegundos, con o sin cortes intermedios.
  - El tráfico de red y el *load average* de esta máquina cumplen el mismo propósito de las instancias de `node`.
  - Los requests recibidos nos sirven para ver si efectivamente hubo un llamado a esta máquina: ya que nuestro escenario envía requests frecuentemente y sin pausa, un corte nos significaría que nunca hubo un llamado y que el solicitante resolvió el pedido por sí mismo (¡con una cache!).

## Estudio 1 - Node Singular

La configuración a analizar consiste de una sola instancia de la máquina de **node**, sin ningún tipo de cache, que recibe todo lo que se le pregunte a la VMSS y, al ingresar un pedido a **/remote**, hace un llamado remoto a la máquina de **gunicorn**.



Hosts al tener sólo una instancia de node

Vamos a analizar una corrida de 4 minutos y 16 segundos, de la cual (sin contar los 5 usuarios creados para el **ping** inicial) se completaron exitosamente 93 de los 726 usuarios creados en el escenario.

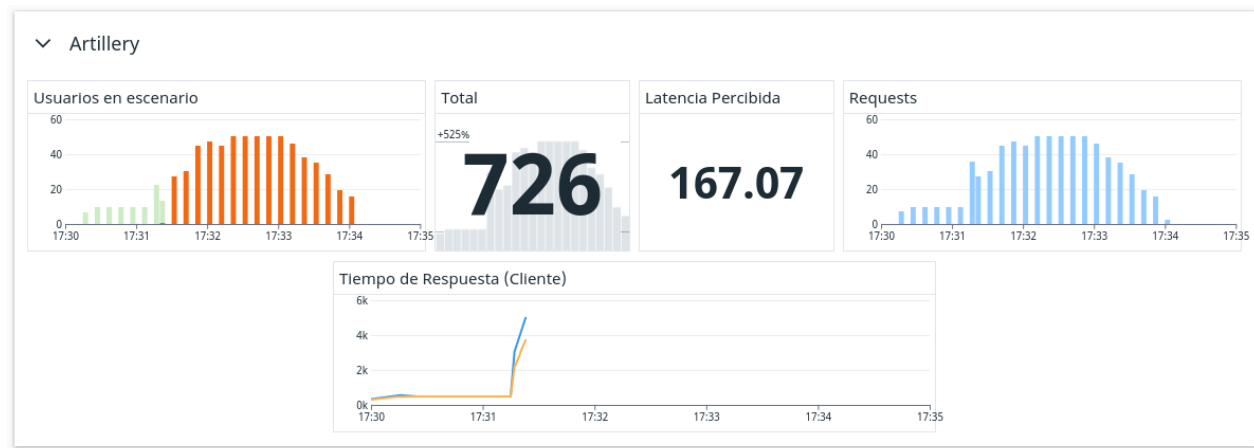
```
$ ./run.sh "/remote"
All VUs finished. Total time: 4 minutes, 16 seconds

-----
Summary report @ 17:34:02(-0300)
-----

errors.ETIMEDOUT: ..... 633
http.codes.200: ..... 93
http.request_rate: ..... 3/sec
http.requests: ..... 726
http.responses: ..... 93
vusers.completed: ..... 93
vusers.created: ..... 726
vusers.failed: ..... 633
```

Los usuarios fallidos son todos por el mismo motivo: **ETIMEDOUT**. Por defecto, **artillery** tiene un tiempo de espera de 10 segundos antes de salir con error. Este número nos parece apropiadamente elegido: si un pedido tarda más de 10 segundos, vamos a tomarlo como fallido, y vamos a considerar que estamos acercándonos al punto de quiebre del sistema.

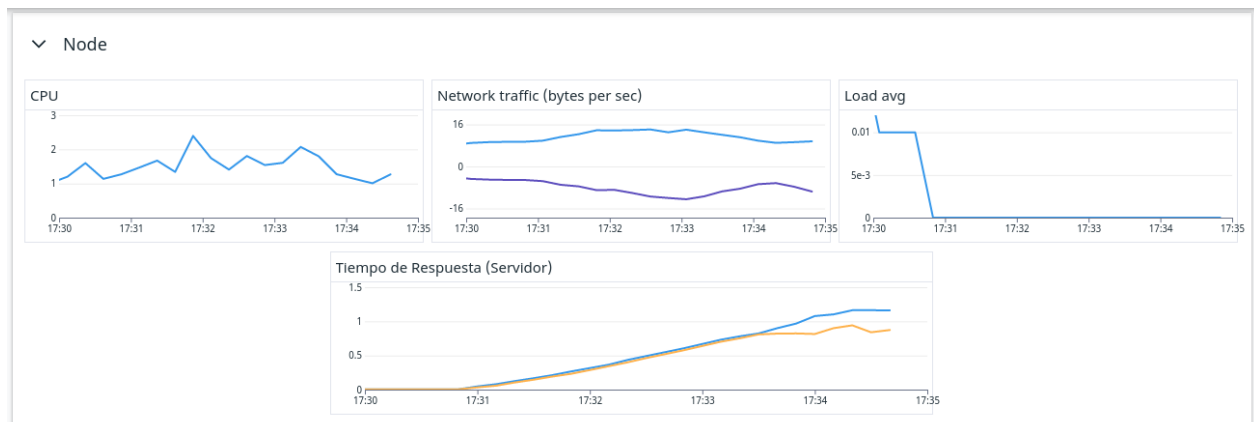
Como aclaración, esto no significa que la instancia de **node** haya dejado de funcionar, sino que simplemente tardó mucho en respondernos. Desde su lado, tranquilamente **node** sigue recibiendo pedidos y llamando al servicio externo. Esto todavía no lo sabemos, pero lo vamos a confirmar más adelante.



Node Singular - Local

Claramente se puede ver que el sistema comienza a fallar en la fase de **RampUp**, de manera casi instantánea: se disparan los tiempos de respuesta, y los usuarios **fallidos** comienzan a superar enteramente a los usuarios **completados**.

En esta fase es en donde apenas empiezan a aumentar la cantidad de requests a más de una por segundo. Es decir: **con sólo una instancia, el sistema no tolera más de 2 usuarios por segundo**. Intentar decir “está alrededor de 1.7 requests por segundo” es un análisis complejo: no tiene sentido en la vida real decir que mandamos 1 request y fracción de otro<sup>2</sup>, es decir no se pueden fraccionar los requests.

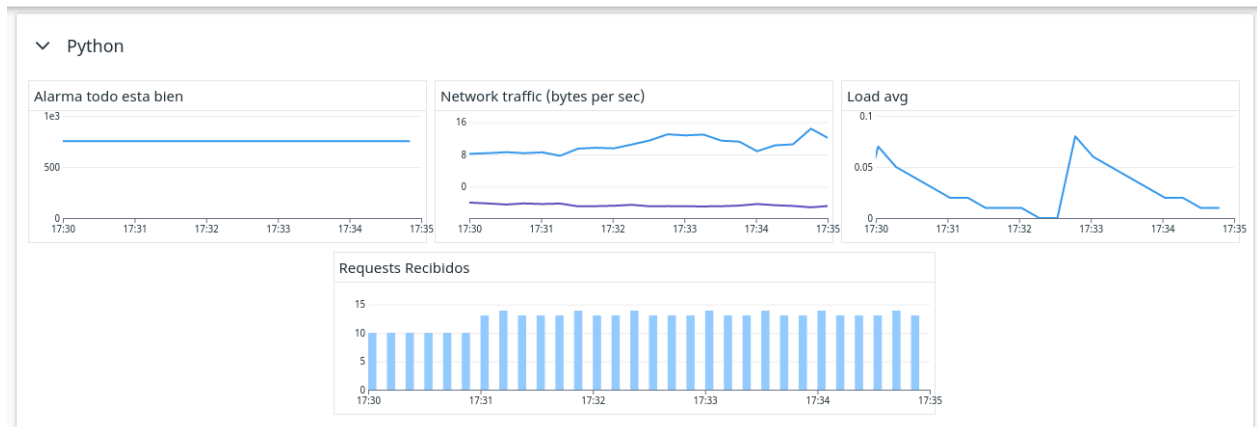


Node Singular - Servidor Node

Desde el punto de vista del servidor, también vemos que el punto de quiebre se encuentra alrededor de las **17:31:30hs**. Es en donde comienza el **RampUp** y se tiene el pico global de uso de CPU.

Lo interesante del tiempo de respuesta (tanto desde el cliente como desde el servidor) es que se puede ver que es lineal en relación al **RampUp**. Comienza constante, y al empezar a recibir más de un usuario, crece linealmente.

<sup>2</sup>Como bien nos enseña el fundador de artillery en la [sección de issues](#)



Node Singular - Servicio Externo

Finalmente, podemos ver que desde el servicio externo todo funciona de manera estable. Se reciben pedidos y siempre se mantuvo constante el tiempo de demora de 750ms.

Sin embargo, algo que se puede notar en el gráfico es que incluso a las 17:35hs el servicio externo sigue procesando pedidos en el patrón **Plain** en vez de **CleanUp**. Pero, conociendo nuestro escenario corrido, sabemos que para esta hora deberíamos estar dejando de recibir requests. Entonces, ¿dónde están los pedidos que faltan?



Node Singular - Pedidos al servicio externo

Si nos fijamos una franja horaria más extensa podemos confirmar que todos los pedidos se completaron, pero en el rango de 10 minutos en vez de 5. Con este dato faltante podemos terminar nuestro rompecabezas y concluir que el cuello de botella esta en el pedido entre **node** y **python**.

Lo que asumimos que sucede es que **node** de manera asíncrona envía todos los pedidos que recibe al servidor externo, mientras que **python** por su lado los va encolando y resolviendo secuencialmente. Por esto mismo, 726 requests completados, a razón de al menos 750ms en dar la respuesta por cada uno resulta en  $((726 * 750ms) / 1000) / 60s$ , que es igual a 9.075 minutos. Esta es la cota inferior de lo que tarda el servicio externo en responder a esa cantidad de requests. Por lo tanto, al sólo tener una instancia de node que se queda esperando la respuesta del servicio externo, para así responderle al cliente, todo el proceso se ve bastante demorado y casi todas las respuestas terminan superando los 10 segundos de *time out* que marcamos.

## Estudio 2 - Node Singular con Redis

En este escenario, analizamos la configuración de una sola máquina de **node**, pero con un cache de Redis intermedio entre **node** y el servicio externo en **python**.

```
# Confirmamos que `/remote/cached` es más rápido que `/remote/`
# Veamos el primer hit y los últimos 4, así ya tenemos la cache en uso
$ for run in {1..15}; do time curl
↳ "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/;" done
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/" 1,086 total
...
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/" 1,086 total
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/" 1,087 total
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/" 1,085 total
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/" 1,083 total
$ for run in {1..15}; do time curl
↳ "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/cached"; done
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/cached" 1,128 total
...
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/cached" 0,334 total
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/cached" 0,330 total
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/cached" 0,331 total
{"id":1}
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/cached" 0,332 total

# Por supuesto, nunca hay que olvidarse de limpiar el cache entre prueba y prueba
curl "http://ladrillo-fdm.eastus.cloudapp.azure.com/remote/cached" -vX DELETE
```

En el Estudio 1 vimos que el cuello de botella se encontraba en el pedido al servicio externo, por lo tanto nuestra hipótesis es que agregando una cache intermedia los tiempos de respuesta van a mejorar sustancialmente.

La prueba de Artillery realizada fue la misma que en el Estudio 1, con las fases de **ping**, **WarmUp**, **RampUp**, **Plain** y **CleanUp**, y usando el endpoint **/remote/cached/**.

Además, cabe aclarar que para estos escenarios utilizamos una configuración de **cacheKeyLength** de 10. Esto define el tamaño de la cache y por tanto la cantidad de keys que va a soportar Redis, por lo que mientras más grande sea, más requests serán necesarias para llenarla. Al ser de tamaño 10, los primeros 10 requests no hacen hit en la cache y siguen de largo hacia el servicio externo, pero a partir del request 11, se empieza a usar la cache.

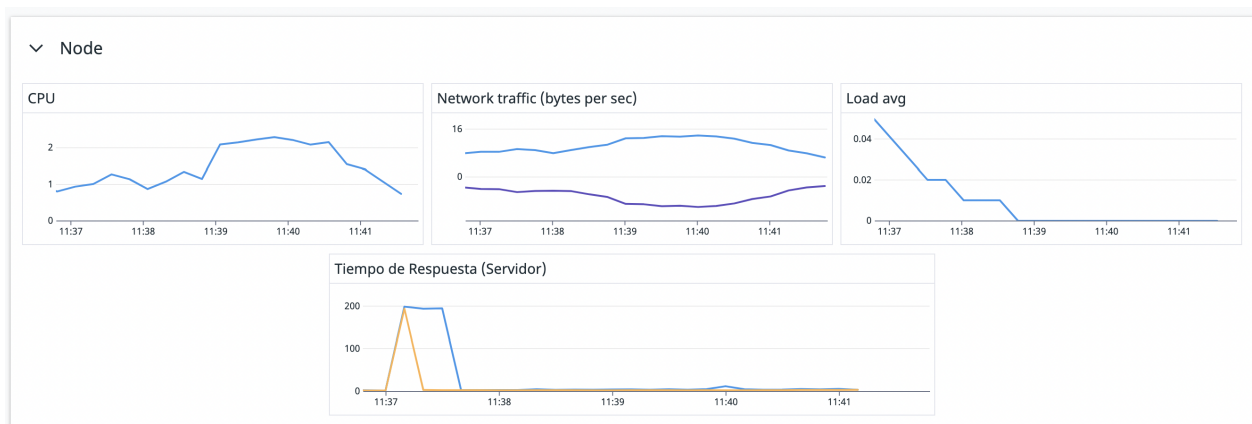




Node Cache - Local

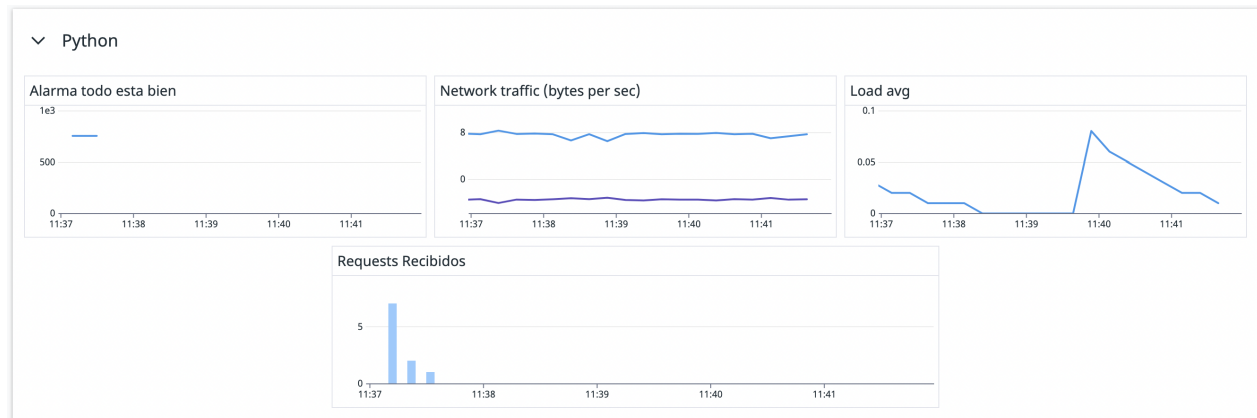
En las métricas del lado de Artillery (usuario), podemos apreciar, en primer lugar, que todos los requests del escenario fueron completados con éxito sin fallas, esto nos da un indicio de que la mejora aplicada fue de gran utilidad.

Si observamos el tiempo de respuesta, vemos cómo hay un pico al inicio y después se descende y se mantiene constante en un nivel muy bajo. El pico se condice con los primeros 10 requests que **node** hace a **python** y no están cacheados aún. El descenso y planicie comienzan partir del siguiente request, donde todos empiezan a ser hit en la caché y no hay necesidad de tener que llegar al servicio externo.



Node Cache - Servidor Node

Observando las métricas del lado del servidor de **node**, vemos que el tiempo de carga promedio y el tiempo de respuesta tienen sus máximos al comienzo, mientras se llena la cache, y luego van en descenso una vez que la cache está completa. Esto es coherente con las métricas de Artillery analizadas previamente.

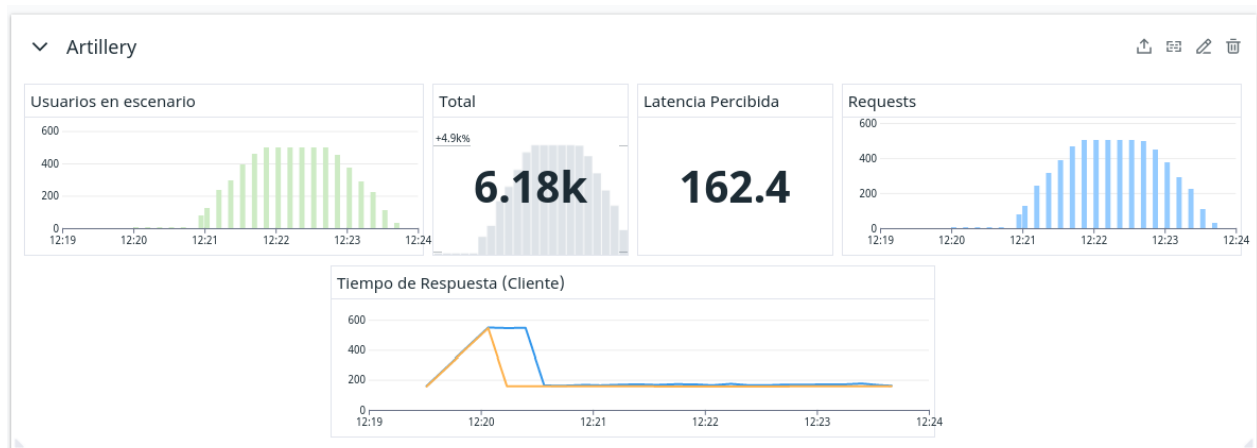


Node Cache - Servicio Externo

Por último, si vemos del lado de **python**, notamos que la cantidad de requests recibidos son 10 en total (7, 2 y 1). Esto confirma el hecho de que estamos usando una cache de tamaño 10 y que una vez llena, todos los requests que saldrían de **node** hacia **python** no se terminan haciendo porque su respuesta ya se encuentra en Redis.

La alarma todo esta bien nos muestra exáctamente lo que dijimos que mostraría: una línea constante con un corte. Atiende los primeros 10 pedidos, y luego deja de llamarse al **sleep** de **python**. Si tuviésemos el ambiente con 2 réplicas, acá veríamos dos líneas entrecortadas.

Viendo que todo funciona correctamente, queremos subir la intensidad: realizamos una segunda prueba (de manera manual), pero con un **RampUp** de hasta 50 usuarios por segundo.

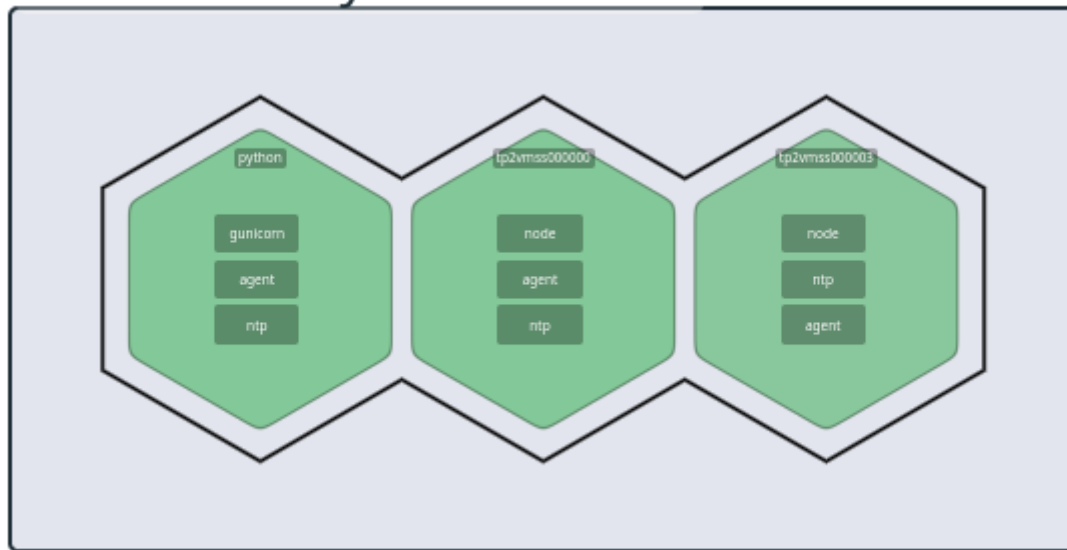


Node Cache - Local

Nuevamente todos los requests son completados satisfactoriamente y se puede concluir que el hecho de agregar una cache intermedia, al menos con el tamaño de cache propuesto (10), mitiga el cuello de botella del caso anterior.

## Estudio 3 - Node Replicado x2

Para este analisis se utilizarán dos instancias de **node**<sup>3</sup> y el endpoint al que llamaremos será nuevamente **/remote**, ya que no queremos utilizar ningún tipo de cache en esta configuración.



Hosts al tener tres instancias de node

Inicialmente lo que esperaríamos ver es un escenario más parecido al caso 1, donde efectivamente veamos que nuevamente se sobrecargue el sistema debido a la ausencia de cache. Sin embargo, en este escenario pretendemos que la falla de los pedidos ocurra más adelante: ya que se introduce una mejora al replicar el servidor de **node**, suponemos que ahora nuestro sistema va a tolerar más pedidos.



Node Replicated - Artillery

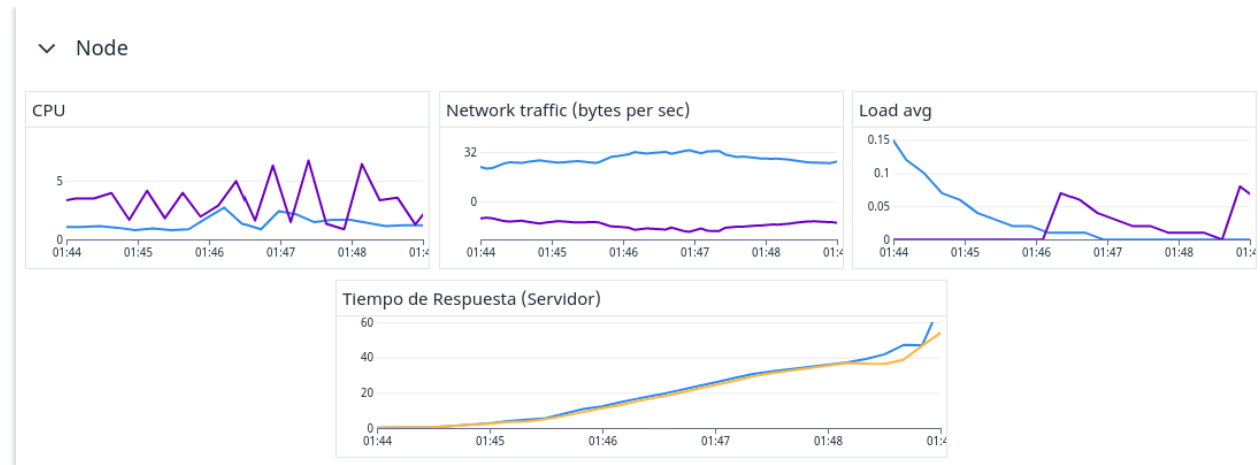
Viendo inicialmente las métricas provistas por artillery, podemos ver que efectivamente se sobrecarga el

<sup>3</sup>Originalmente teníamos pensado hacer un ambiente con 3 réplicas, pero nuestra suscripción actual de Azure solo nos permite tener 4 CPUs virtuales a la vez, y eso lo tenemos al límite: **mgmt**, **python**, **node1**, **node2**.

Quota name	Region	Subscription	Current Usage	Adjustable
Usage at or near quota (2)				
<input type="checkbox"/> Total Regional vCPUs	East US	Azure subscription 1	100% 4 of 4	Yes
<input type="checkbox"/> Standard B5 Family vCPUs	East US	Azure subscription 1	100% 4 of 4	Yes

sistema y que esto sucede pasado el minuto y medio (el punto de quiebre del estudio 1), y podemos confirmar que la cantidad de usuarios completados es superior al escenario 1.

A su vez, en los tiempos de respuesta podemos ver cómo comienza a aumentar pronunciadamente en el mismo momento que comienza a sobrecargarse el sistema con usuarios.

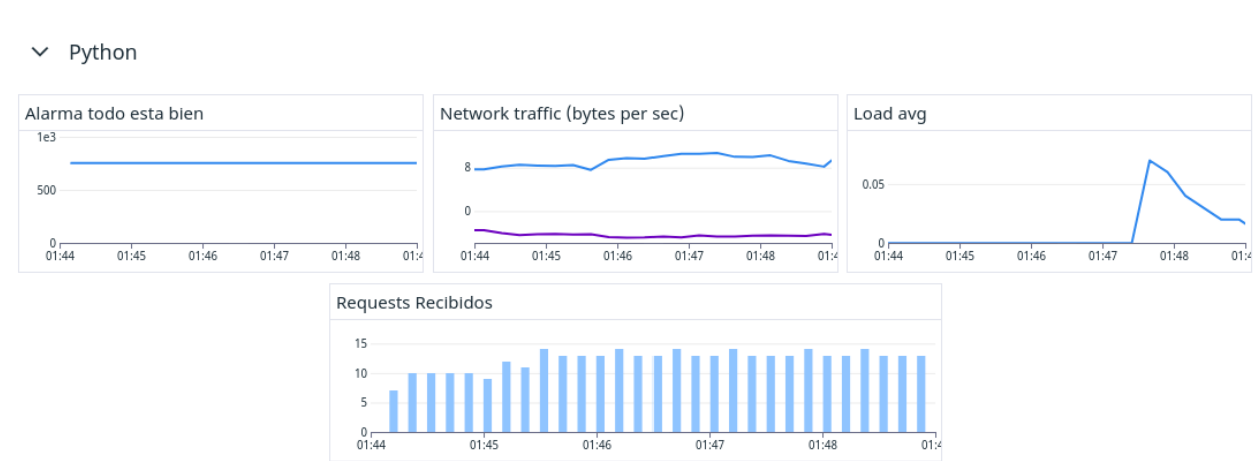


Node Replicated - Node

Para este caso de estudio modificamos el dashboard para poder graficar ambas VMs por separado, es por eso que ahora tanto el CPU como el *load average* tienen dos líneas, una por cada instancia. Es con este nuevo gráfico que podemos darnos una idea de como está el *load balancer* delegando el trabajo.

Desde el punto de vista del servidor, también vemos que el punto de quiebre se encuentra aproximadamente en el momento en que comienza el **RampUp**.

Nuevamente el tiempo de respuesta del servidor tiene una forma lineal en relación al **RampUp**. Comienza constante, y al empezar a recibir más de un usuario, crece linealmente.



Node Replicated - Servicio Externo

En esta imagen, podemos apreciar que en el servicio externo todo funciona correctamente. Se recibieron todos los requests, se manejaron correctamente, y siempre se mantuvo constante el tiempo de demora de 750ms. De la misma manera que lo vimos en el escenario uno, podemos ver como, incluso pasados los 5 minutos de corrida, sigue habiendo pedidos para procesar.

Si bien replicar el servidor de node no es la solución al problema analizado en este trabajo, sí tiene utilidades. Supongamos el sistema con cache que mostramos en el ítem anterior, si a ese sistema se le enviaran muchísimas requests por segundo, es probable que el mismo colapse pero no por la cache, sino porque la única replica de node que funciona allí no puede manejar tantos requests (antes de enviárselos al servicio externo). En ese caso, sí sería útil tener más réplicas de node para poder distribuir la carga entre ellas antes de enviar sus respectivos requests al servicio externo con cache en el medio.

## Conclusiones

En el presente trabajo práctico pudimos comprender y poner en práctica diversos conceptos y herramientas como fueron servicios cloud, herramientas de monitoreo cloud, ansible, una gama de servicios de Azure, y demás.

Nos fue posible entender qué se nos estaba pidiendo y cómo lograrlo, además de formular hipótesis sobre los resultados que creímos que íbamos a obtener. Luego tuvimos la oportunidad de verificar dichas hipótesis contra las métricas que obtuvimos, y conseguimos comprobar algunas y descartar otras.

Pudimos comprender cómo resolver el problema planteado, lo cual resultó muy provechoso para entender cómo funcionan los servicios cloud y cómo se pueden utilizar para resolver problemas de esta naturaleza. Vimos que si bien uno podría suponer que agregar réplicas de node parecía la forma de llevar a cabo el problema propuesto, lo que había que hacer realmente era determinar dónde estaba efectivamente el problema, y buscar soluciones a éste, como por ejemplo introducir una cache.