

# Performance

Pruebas de rendimiento en Liferay

**Material:**

[https://dmcisneros.github.io/lug\\_pruebas\\_carga/](https://dmcisneros.github.io/lug_pruebas_carga/)



PLANIFICA



DEFINE



EJECUTA Y  
ANALIZA



David Vega Perea (@davidsrules7) - Minsait  
Daniel Martínez Cisneros (@dmcisneros) - Everis





# Nuestro Objetivo

El objetivo de éste repositorio es ver como realizar unas pruebas de carga coherentes sobre un entorno Liferay, nos servirán para:

- Demostrar que el sistema cumple los criterios de rendimiento.
- Validar y verificar atributos de la calidad del sistema: escalabilidad, fiabilidad, uso de los recursos.
- Medir qué partes del sistema o de carga de trabajo provocan que el conjunto rinda mal.

# PRUEBAS DE RENDIMIENTO

3

PLANIFICA Y DISEÑA

01

ANALIZA Y MIDE  
RESULTADOS

03

02

04

ATACA LOS PUNTOS  
DEBILES DE TU  
ARQUITECTURA

AJUSTA Y MEJORA  
EL RENDIMIENTO

# JMeter – Pruebas de carga

Utilizaremos Jmeter por ser suficiente para las pruebas que necesitamos, ser la más extendida y por ser opensource.



## Tipos de pruebas:

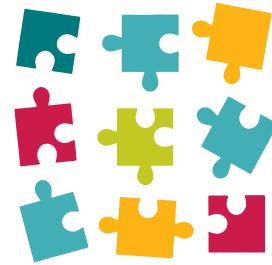
- **Pruebas de carga:** Las pruebas de carga son un tipo de prueba de rendimiento del sistema. Con ellas observamos la respuesta de la aplicación ante un determinado número de peticiones.
- **Pruebas de estrés:** Este es otro tipo de prueba de rendimiento del sistema. El objetivo de estas pruebas es someter al software a situaciones extremas, intentar que el sistema se caiga, para ver cómo se comporta, si es capaz de recuperarse.



Existen multitud de herramientas que nos facilitan la posibilidad de lanzar pruebas de carga:

- |                 |                   |
|-----------------|-------------------|
| - WebLOAD       | - LoadNinja       |
| - SmartMeter.io | - Tricentis Flood |
| - LoadView      | - LoadUI NG Pro   |
| - LoadRunner    | - Appvance        |
| - NeoLoad       | - LoadComplete    |
| • .....         |                   |

# JMETER - Elementos



1

**Test Plan:** Representa la raíz del plan de pruebas.

2

**Thread Group:** Representa un grupo de usuarios, cada thread es un usuario virtual.

3

**Controllers:** Samplers, Logic Controlers, Config Element, Assertion, Listeners, Timer, Pre-Processor element, Post-Processor element.

4

**Plugins de Jmeter:**

- 3 Basic Graphs: <https://jmeter-plugins.org/wiki/ResponseTimesOverTime/>
- 5 Additional Graphs: <https://jmeter-plugins.org/wiki/ResponseCodesPerSecond/>

# JMETER – Pruebas definidas

Para la simulación de usuarios virtuales haremos uso de las opciones definidas a continuación que nos facilitan los plugins instalados:

- **jp@gc Throughput Shaping timer:** Definiremos el número de peticiones que se realizaran por segundo (RPS)
- **jp@gc Ultimate Thread Group:** Se indicarán el número de hilos (usuarios)

Dichas gráficas deben ser semejantes y los hilos que se indiquen en la opción de thread group se deben calcular con la siguiente fórmula:

$$\text{RPS} * \langle\text{max response time}\rangle / 1000$$

En nuestro caso simularemos 200 usuarios concurrentes con una rampa de subida y otra de bajada para hacerla progresiva, para el cálculo de RPS e hilos tendremos como resultado de la fórmula anterior.

$$\text{Hilos} = 80 * 2500/1000 = 200$$

# JMETER – Pruebas definidas

Quedando las gráficas de la siguiente forma:

- jp@gc Throughput Shaping timer:



- jp@gc Ultimate Thread Group:



¡NO LANZAR LAS PRUEBAS NUNCA DESDE LA MISMA MÁQUINA EN LA QUE ESTÉ EL SERVICIO!

# Lanzar pruebas y monitorizar comportamiento

Vamos a analizar el comportamiento del portal de liferay con 10 contenidos web, 5 con una estructura con plantilla no cacheable y otros 5 con una estructura con plantilla cacheable. Del mismo modo, se estudiará la respuesta de un portlet a medida según utilice cache o no.

## Pruebas previstas:

- **Página con un visor de contenidos:**

- **Url:** /01\_test\_lug\_visor
- **Descripción:** Al ser el visor por defecto con un contenido cacheable debería ser el resultado más optimo de las pruebas realizadas

- **Página con un publicador de contenidos con contenidos web con plantilla sin cachear:**

- **Url:** /02\_test\_lug\_publicador\_no\_cache
- **Descripción:** Al no usar la caché de liferay tendrá que procesar la plantilla del contenido cada vez que realice su renderizado con el coste computacional correspondiente.

- **Página con un publicador de contenidos con contenidos web con plantilla cacheada:**

- **Url:** /03\_test\_lug\_publicador\_cache
- **Descripción:** En la primera carga el resultado será semejante a la página anterior pero en posteriores los tiempos de respuesta serán mejores al estar cacheada la plantilla.

# Lanzar pruebas y monitorizar comportamiento

Vamos a analizar el comportamiento del portal de liferay con 10 contenidos web, 5 con una estructura con plantilla no cacheable y otros 5 con una estructura con plantilla cacheable. Del mismo modo, se estudiará la respuesta de un portlet a medida según utilice cache o no.

## Pruebas previstas:

### •Página con un portlet a medida sin hacer uso de cache:

- **Url:** /04\_test\_lug\_custom\_module\_no\_cache
- **Descripción:** Al tener un portlet a medida que cada vez que se renderiza ejecuta peticiones a API externa y posteriormente un tratamiento considerable de datos, se tendrán tiempos de respuestas muy altos cada vez que se acceda al mismo.

### •Página con un portlet a medida haciendo uso de cache:

- **Url:** /05\_test\_lug\_custom\_module\_cache?cache=true
- **Descripción:** En la primera carga el resultado será semejante a la página anterior pero en posteriores los tiempos de respuesta serán mejores al estar cacheada la llamada a la API y gran parte del tratamiento de datos del módulo a medida.

# Antes de comenzar...

Antes de lanzar las pruebas deberíamos monitorizar el comportamiento de nuestra arquitectura con herramientas como jvisualvm, jmc ó jconsole. Puntos de interés dentro de las métricas que podemos observar:

## Consumo de CPU, comportamiento de JVM, nº de hilos, etc...



# Antes de comenzar...

En la sección Mbeans podemos ver datos interesantes como son:

## Ehcach

Las caches disponibles de liferay y su comportamiento

# Antes de comenzar...

En la sección Mbeans podemos ver datos interesantes como son:

## vPool Hikari

Pool de conexiones usado por defecto en Liferay

The screenshot shows the JConsole interface with the following details:

- Start Page**, **JMX 10.98.134.16:8000 (pid 6948)**, **Tomcat (pid 55247)** tabs.
- MBeans** tab selected in the top navigation bar.
- Tomcat (pid 55247)** selected in the left tree view.
- MBeans Browser** panel on the right.
- MBeans** tree view:
  - Catalina
  - JMImplementation
  - Users
  - com.liferay.portal.messaging
  - com.liferay.portal.monitoring
  - com.sun.management
  - com.zaxxer.hikari
    - Pool (HikariPool-1)
    - Pool (HikariPool-2)
    - PoolConfig (HikariPool-1)
    - PoolConfig (HikariPool-2)
  - java.lang
  - java.nio
  - java.util.logging
  - net.sf.ehcache
    - Cache
- Attributes** tab selected in the MBeans browser header.
- Attribute values** table:

Name	Value
ActiveConnections	1
IdleConnections	10
ThreadsAwaitingConnection	0
TotalConnections	11

# Lanzamientos de Pruebas Elementos OOTB

13

## Prueba #1

**Página con un visor de contenidos:**

/01\_test\_lug\_visor

**Página con un visor de contenidos con contenido web con plantilla cacheada:**

/02\_test\_lug\_publicador\_no\_cache

**Página con un visor de contenidos con contenido web con plantilla sin cachear:**

/03\_test\_lug\_publicador\_cache

# Lanzamientos de Pruebas Elementos OOTB

14

## Prueba #1

### Resultado:

Se puede observar como solo han sido cacheados 5 elementos en memoria.

Attributes	Operations	Notifications	Metadata
<b>Attribute values</b>			
<hr/>			
Name			Value
AssociatedCacheName			com.liferay.journal.util.JournalContent
CacheHitPercentage			0.5241464341268647
CacheHits			73120
CacheMissPercentage			0.4758535658731353
CacheMisses			66383
DiskStoreObjectCount			0
InMemoryHitPercentage			0.5241464341268647
InMemoryHits			73120
InMemoryMisses			66383
MemoryStoreObjectCount			5
ObjectCount			5
OffHeapHitPercentage			0.0
OffHeapHits			0
OffHeapMisses			0
OffHeapStoreObjectCount			0
OnDiskHitPercentage			0.0
OnDiskHits			0
OnDiskMisses			0
WriterMaxQueueSize			0
WriterQueueLength			0

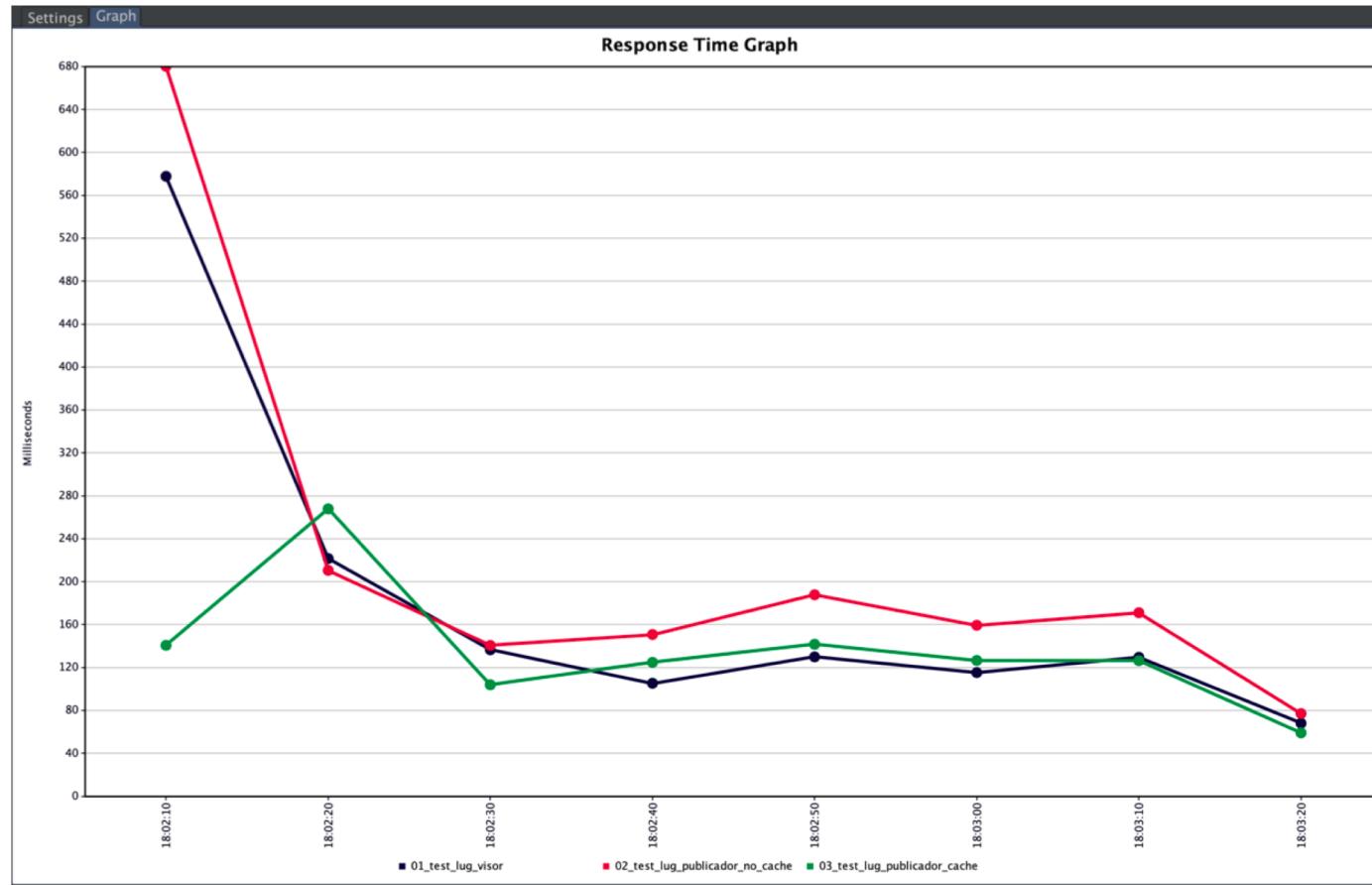
# Lanzamientos de Pruebas Elementos OOTB

15

## Prueba #1

### Resultado:

La página que visualiza los elementos de la plantilla no cacheable tiene unos tiempos de respuesta peores que la cacheada (Verde vs Rojo), en este caso solo tenemos 10 elementos en total pero cuando un sistema empieza a crecer en varios miles de contenidos se puede apreciar esta diferencia de forma más notable.



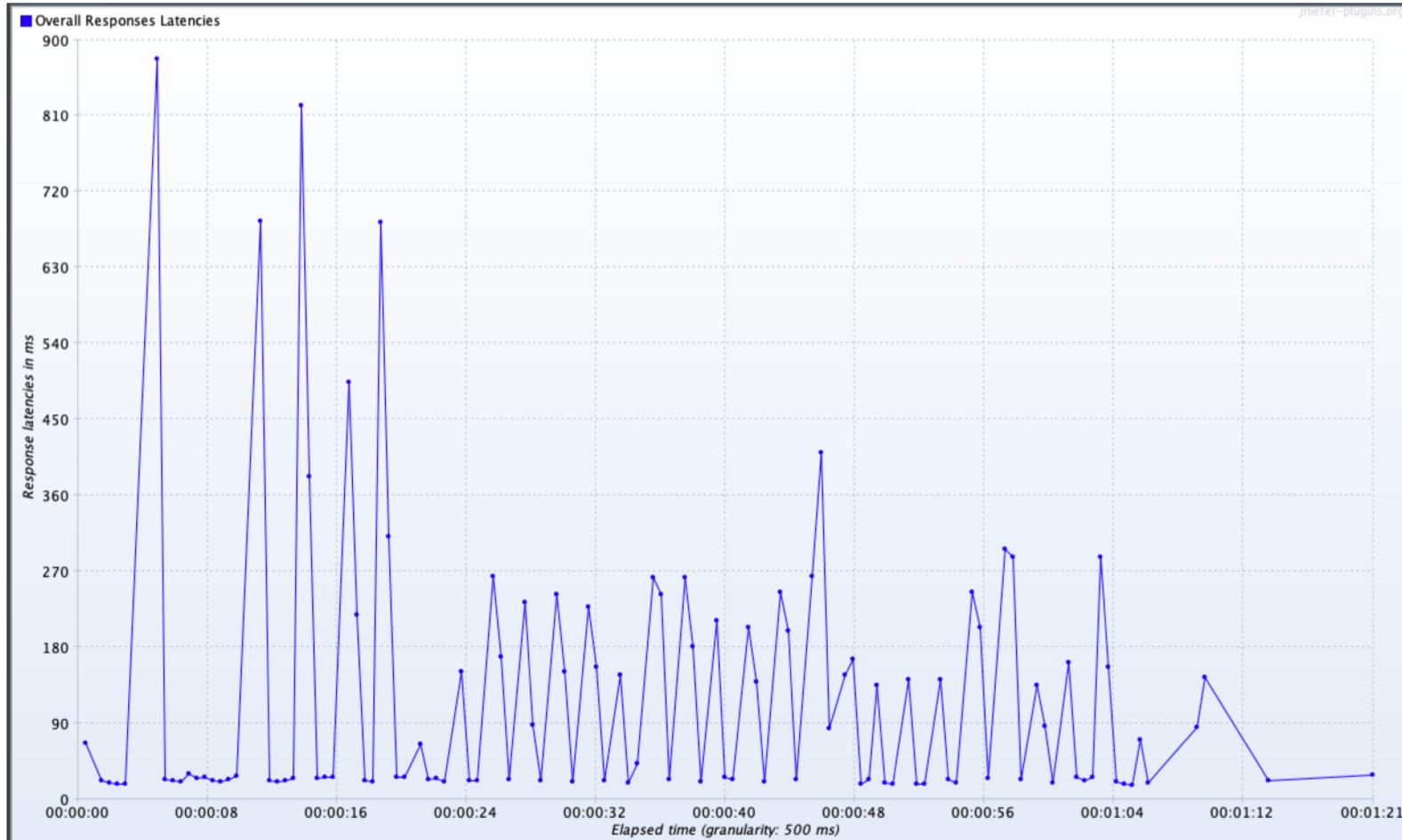
# Lanzamientos de Pruebas Elementos OOTB

16

## Prueba #1

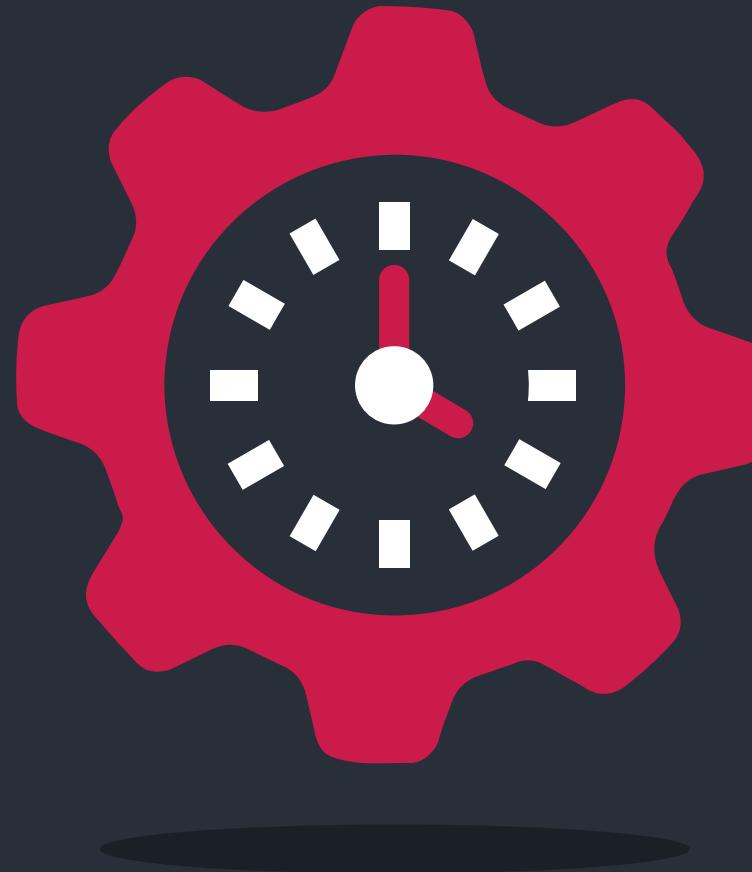
### Resultado:

De la misma forma se observan picos constantes de latencias altas.



# ¡Lo vemos!

Lanzamientos de Pruebas Elementos OOTB – Prueba #1



# Lanzamientos de Pruebas Elementos OOTB

18

## Prueba #2

Se cambiará la configuración de la plantilla que no era cacheable, ahora se pondrá cacheable y se ejecutarán las mismas pruebas de carga de Pruebas Elementos OOTB #1 .

### Resultado:

Se observa como se cachean los 10 contenidos.

MBeans	Attributes	Operations	Notifications	Metadata
Catalina JMImplementation Users com.liferay.portal.messaging com.liferay.portal.monitoring com.sun.management com.zaxxer.hikari Pool (HikariPool-1) Pool (HikariPool-2) PoolConfig (HikariPool-1) PoolConfig (HikariPool-2) java.lang java.nio java.util.logging net.sf.ehcache Cache CacheConfiguration CacheManager CacheStatistics MULTI_VM_PORTAL_CAC com.liferay.dynamic com.liferay.dynamic com.liferay.journal.i com.liferay.portal.k com.liferay.portal.k	Attribute values			

The table displays MBean attributes for various components. A red box highlights the 'MemoryStoreObjectCount' and 'ObjectCount' rows under the 'com.liferay.journal.util.JournalContent' attribute. Both values are listed as 10.

Name	Value
AssociatedCacheName	com.liferay.journal.util.JournalContent
CacheHitPercentage	0.5251512818483368
CacheHits	72985
CacheMissPercentage	0.4748487181516632
CacheMisses	65994
DiskStoreObjectCount	0
InMemoryHitPercentage	0.5251512818483368
InMemoryHits	72985
InMemoryMisses	65994
MemoryStoreObjectCount	10
ObjectCount	10
OffHeapHitPercentage	0.0
OffHeapHits	0
OffHeapMisses	0
OffHeapStoreObjectCount	0
OnDiskHitPercentage	0.0
OnDiskHits	0
OnDiskMisses	0
WriterMaxQueueSize	0
WriterQueueLength	0

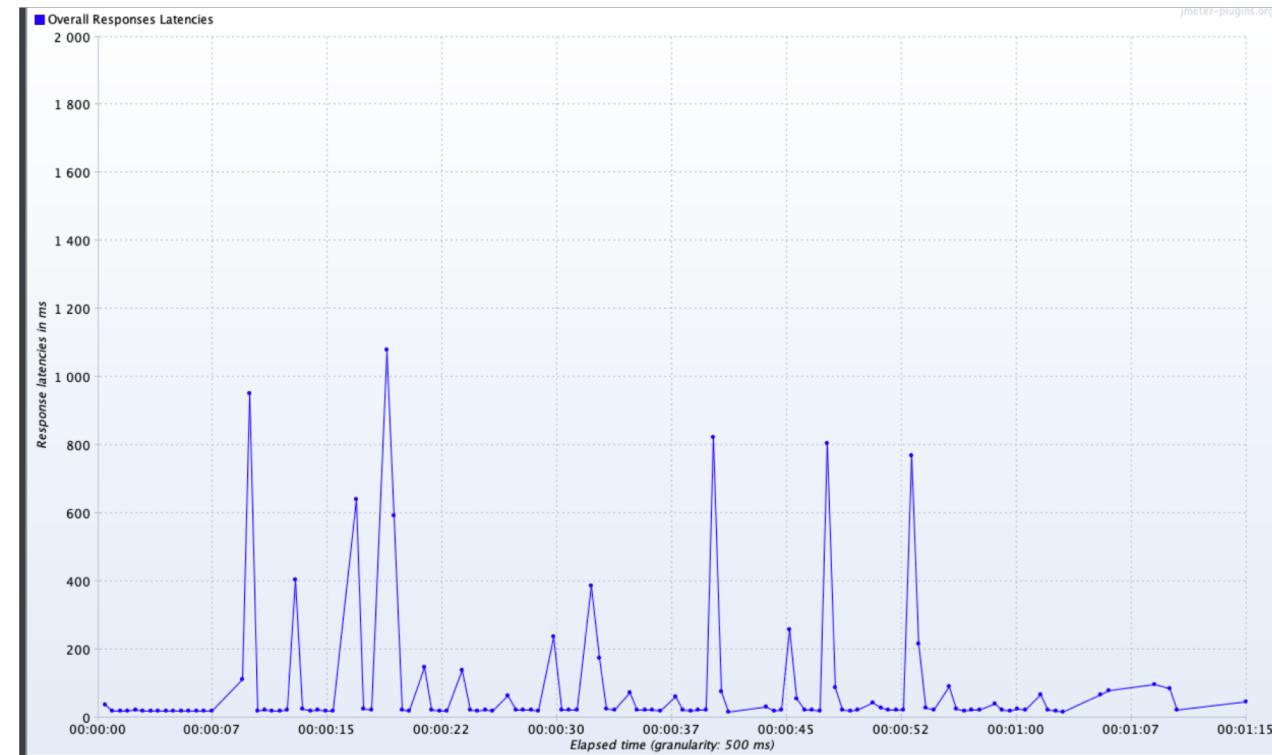
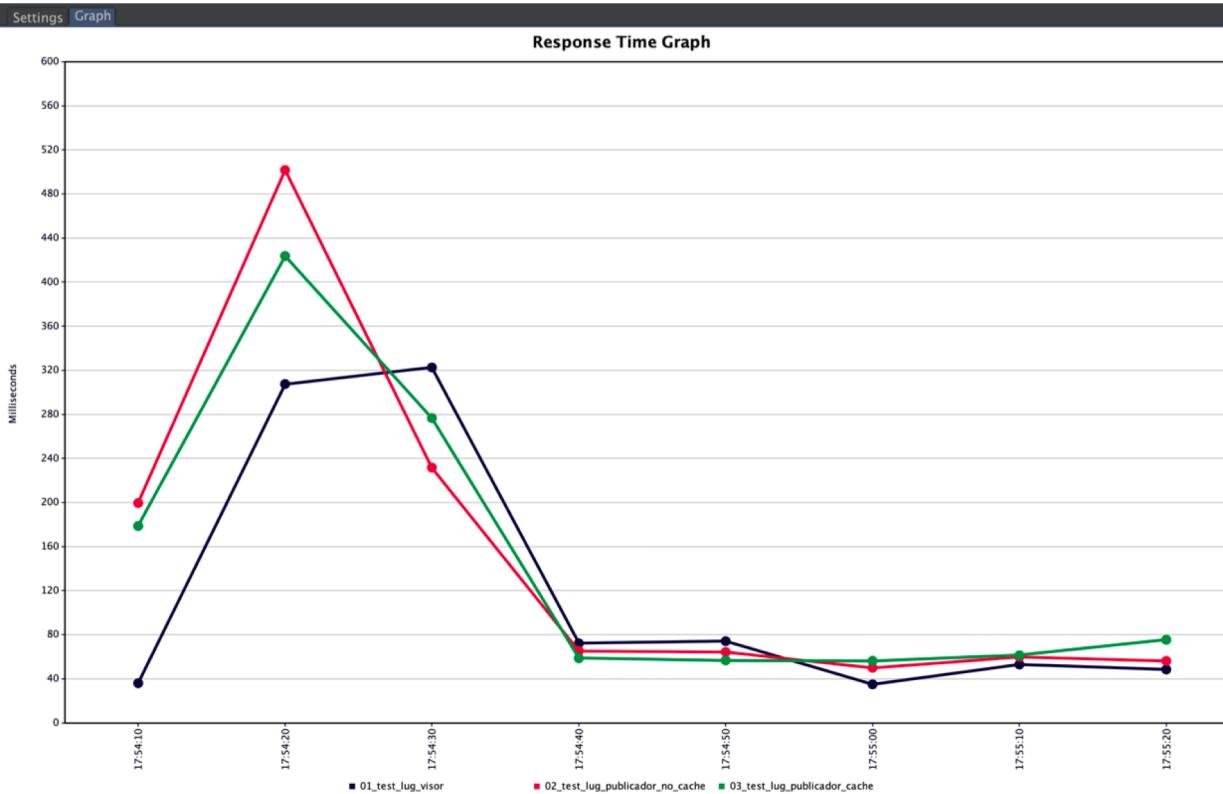
# Lanzamientos de Pruebas Elementos OOTB

19

## Prueba #2

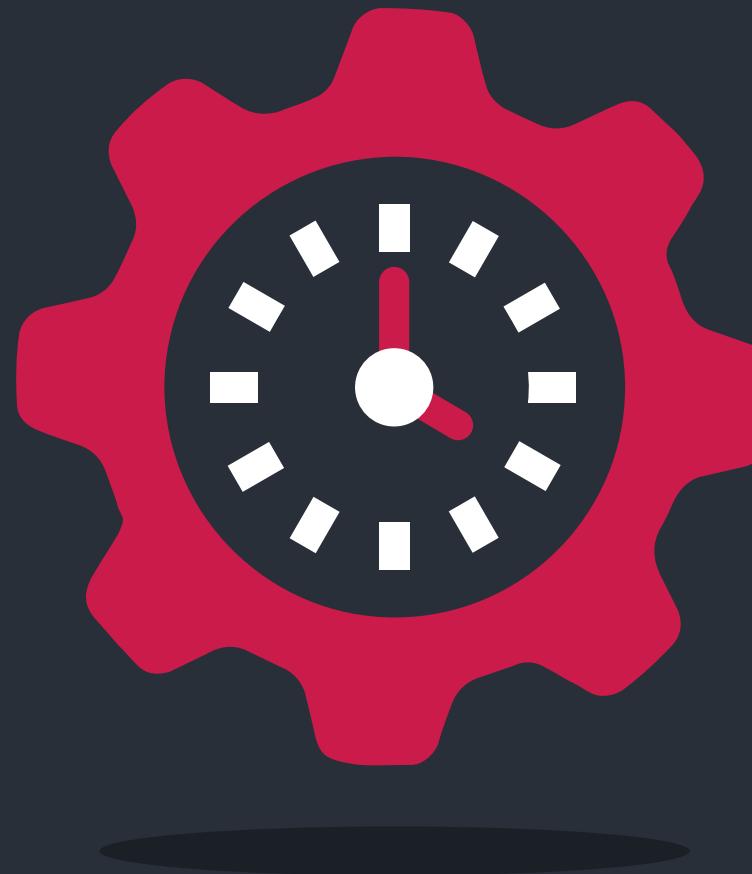
### Resultado:

Las páginas `/02_test_lug_publicador_no_cache` y `/03_test_lug_publicador_cache` ahora muestran resultados semejantes teniendo tiempos de latencia medios más bajos que los resultados anteriores, observándose que en solo 5 contenidos con plantillas no cacheadas anteriormente se degrada la respuesta.



# ¡Lo vemos!

Lanzamientos de Pruebas Elementos OOTB – Prueba #2



# Lanzamientos de Pruebas Módulo Ad hoc

21

## Prueba #1

Página con un portlet a medida sin hacer uso de cache:

`/04_test_lug_custom_module_no_cache`

Para ofrecer una comparación entre los elementos OOTB y un módulo Ad hoc mantendremos las 3 pruebas realizadas sobre elementos OOTB de Liferay

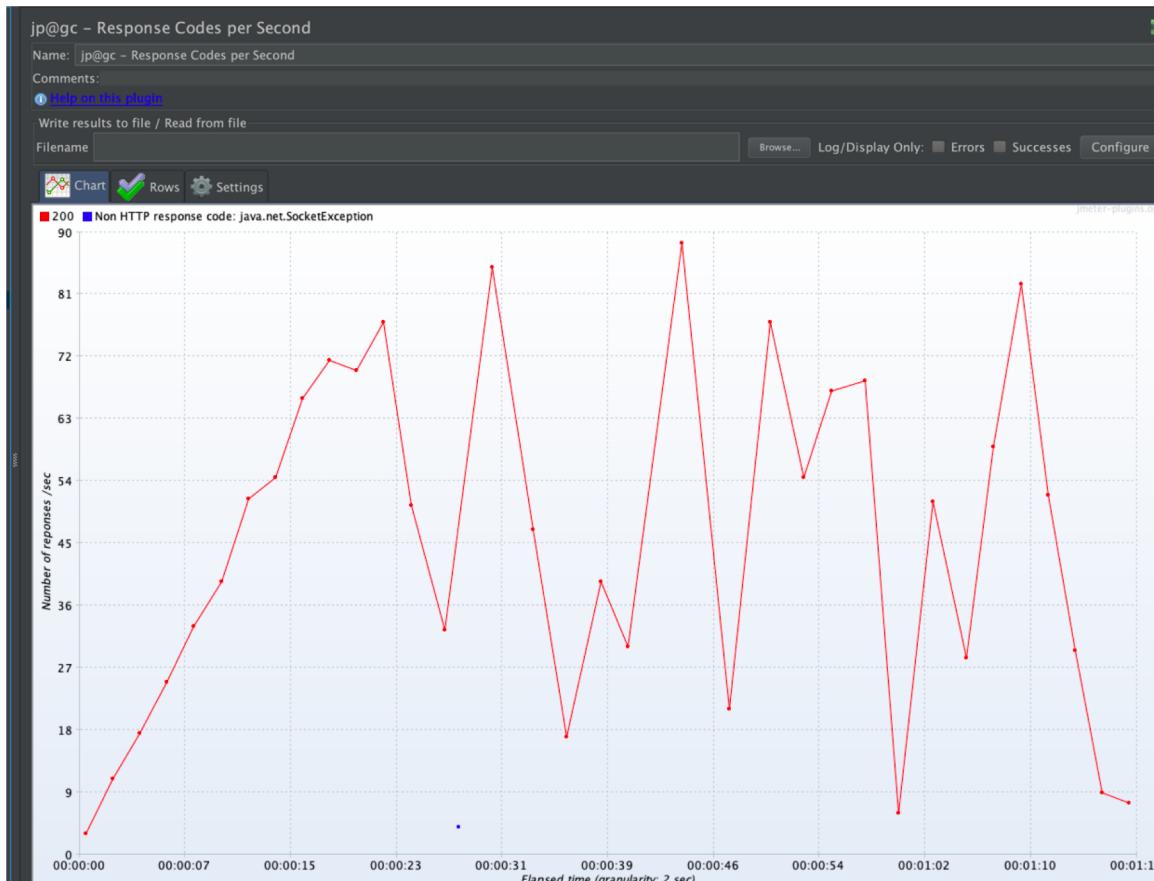
(`/01_test_lug_visor`, `/02_test_lug_publicador_no_cache` y `/03_test_lug_publicador_cache`).

# Lanzamientos de Pruebas Módulo Ad hoc

## Prueba #1

### Resultado:

Se puede observar que no se puede mantener el nivel de respuestas por segundo pretendido (80), sino que se produce un colapso debido al coste que conlleva cada una de las peticiones. Ya que al no cachearse los datos a mostrar, por cada petición se realiza una comunicación de red con una API externa, así como el tratamiento de los datos recibidos.

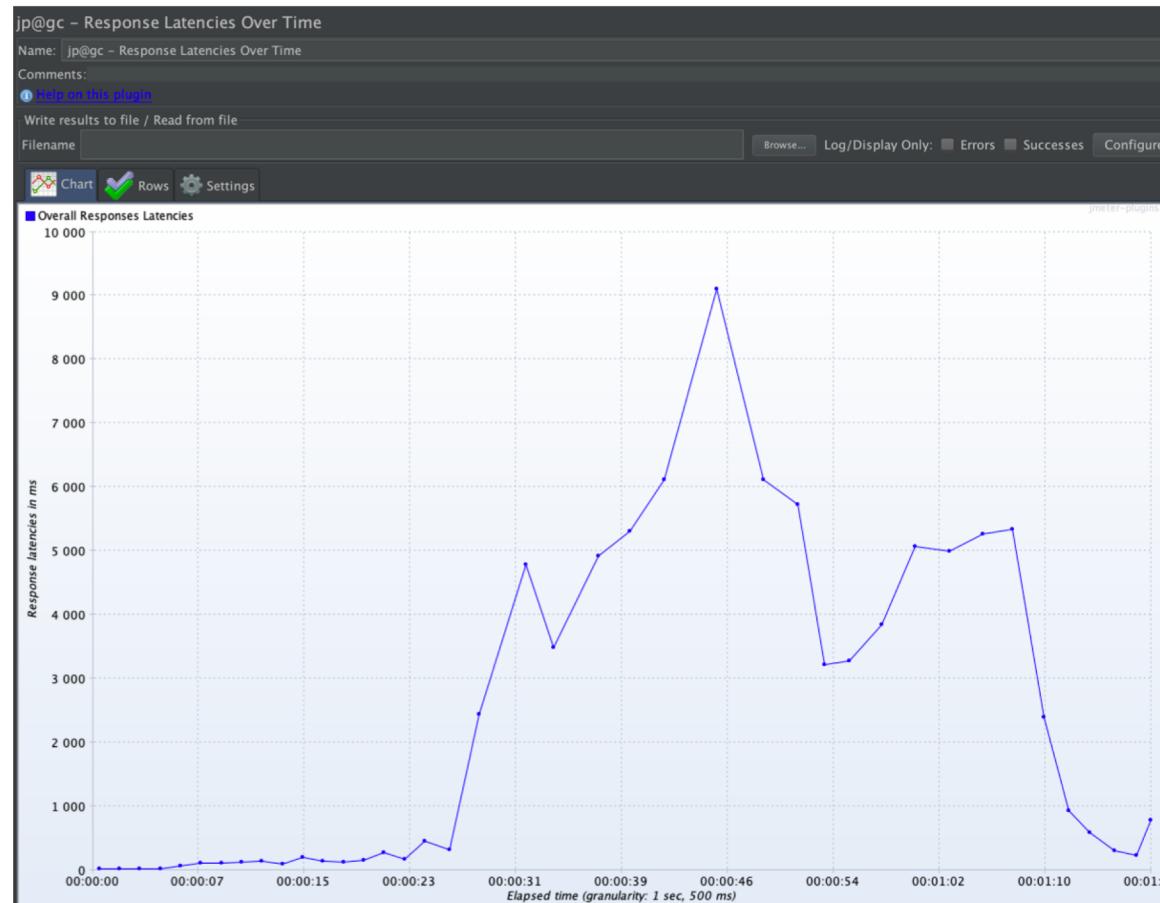


# Lanzamientos de Pruebas Módulo Ad hoc

## Prueba #1

### Resultado:

Debido a los altos tiempos de ejecución que implica una comunicación por red, al no utilizar caché se obtienen latencias altas, donde tras el colapso y acumulación de peticiones aumentan de forma drástica llegando a latencias de 9 segundos.

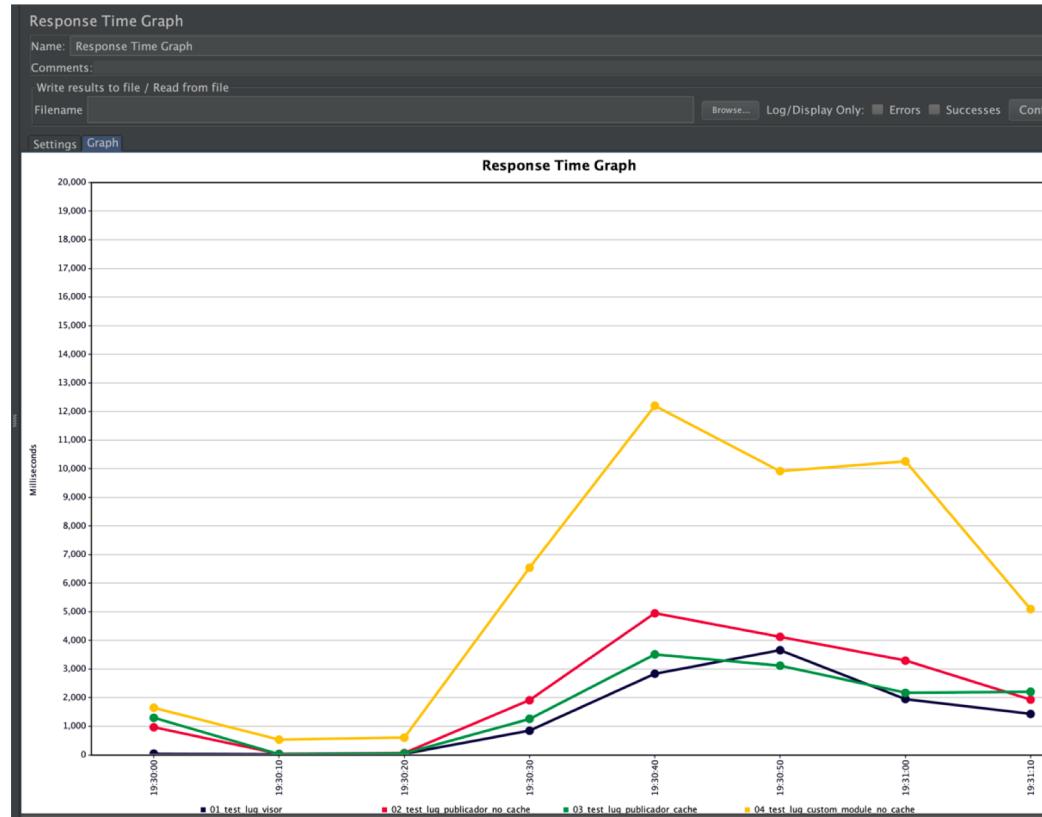


# Lanzamientos de Pruebas Módulo Ad hoc

## Prueba #1

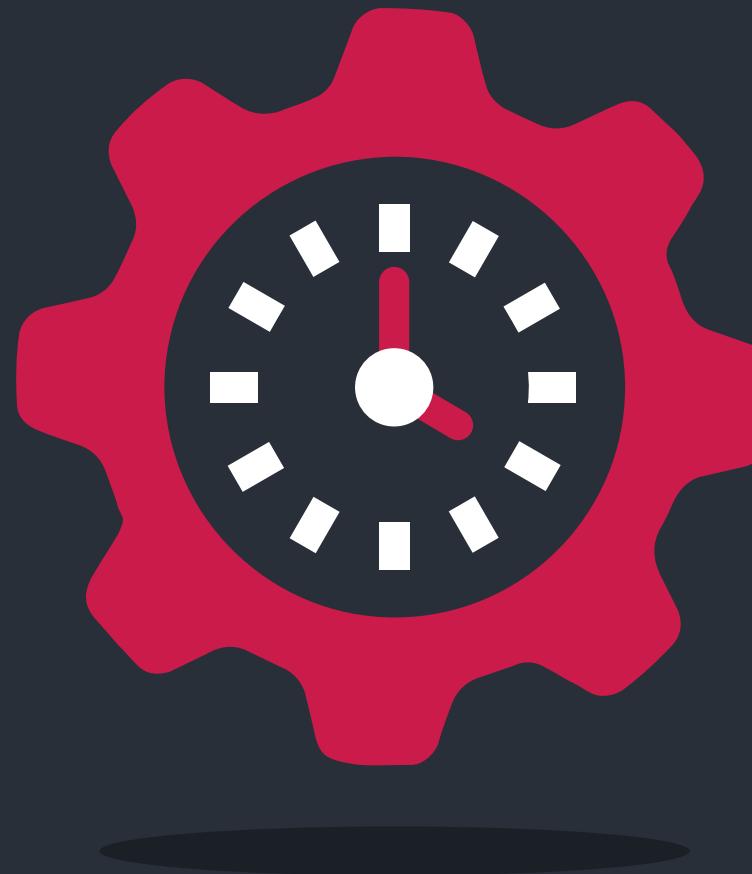
### Resultado:

Si comparamos los tiempos de respuesta obtenidos en esta página con las 3 que contienen elementos OOTB, observamos que siempre se encuentra por encima. Incluso podemos observar como llegados al punto en el que empieza el colapso, no sólo empeora los tiempos de respuestas de la página con el módulo Ad hoc, sino que también degrada el rendimiento en las páginas restantes.



# ¡Lo vemos!

Lanzamientos de Pruebas Módulo Ad hoc – Prueba #1



# Lanzamientos de Pruebas Módulo Ad hoc

26

## Prueba #2

Página con un portlet a medida haciendo uso de cache:

/05\_test\_lug\_custom\_module\_cache?cache=true

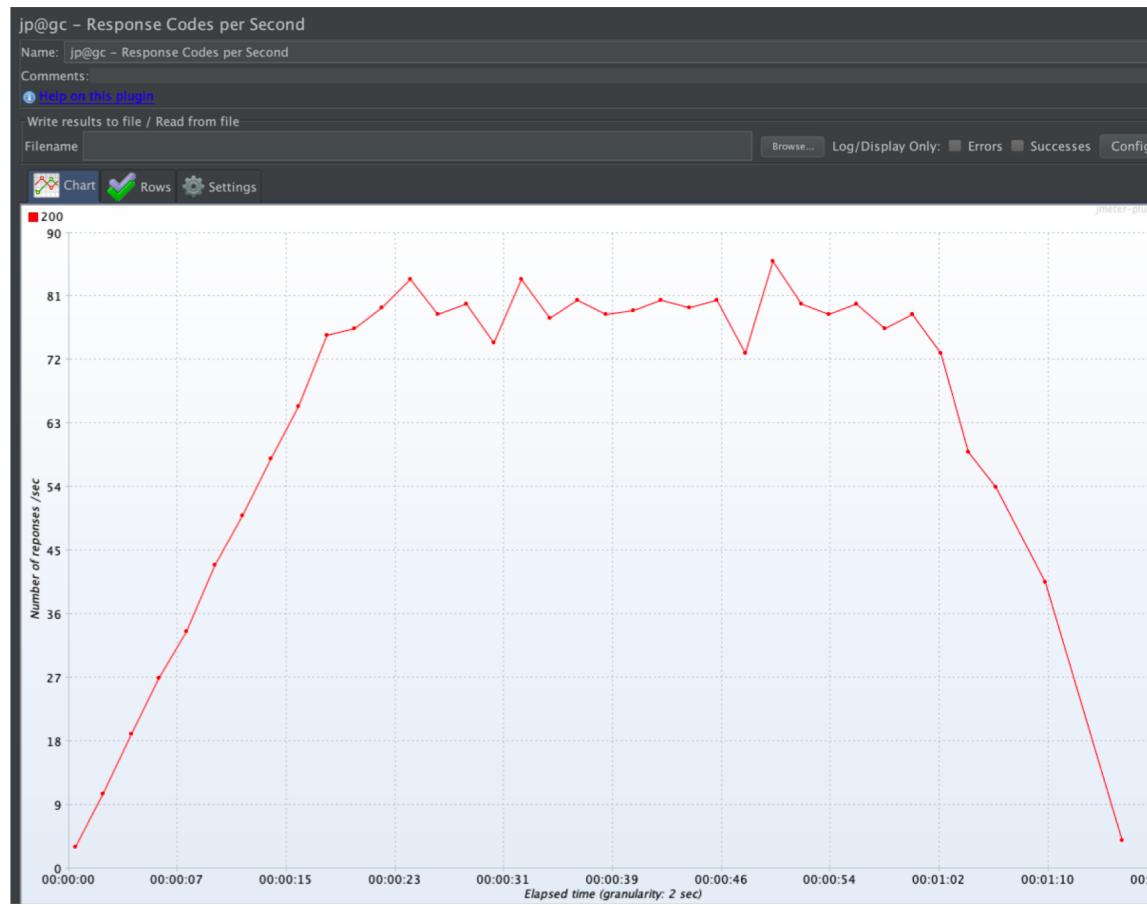
# Lanzamientos de Pruebas Módulo Ad hoc

27

## Prueba #2

### Resultado:

Se puede observar cómo en este caso sí se mantiene el nivel de respuestas por segundo pretendido (80). Ya que al cachearse los datos a mostrar, el 100% del coste de la operación se realiza en la primera petición mientras que en las siguientes se reduce considerablemente el coste computacional.



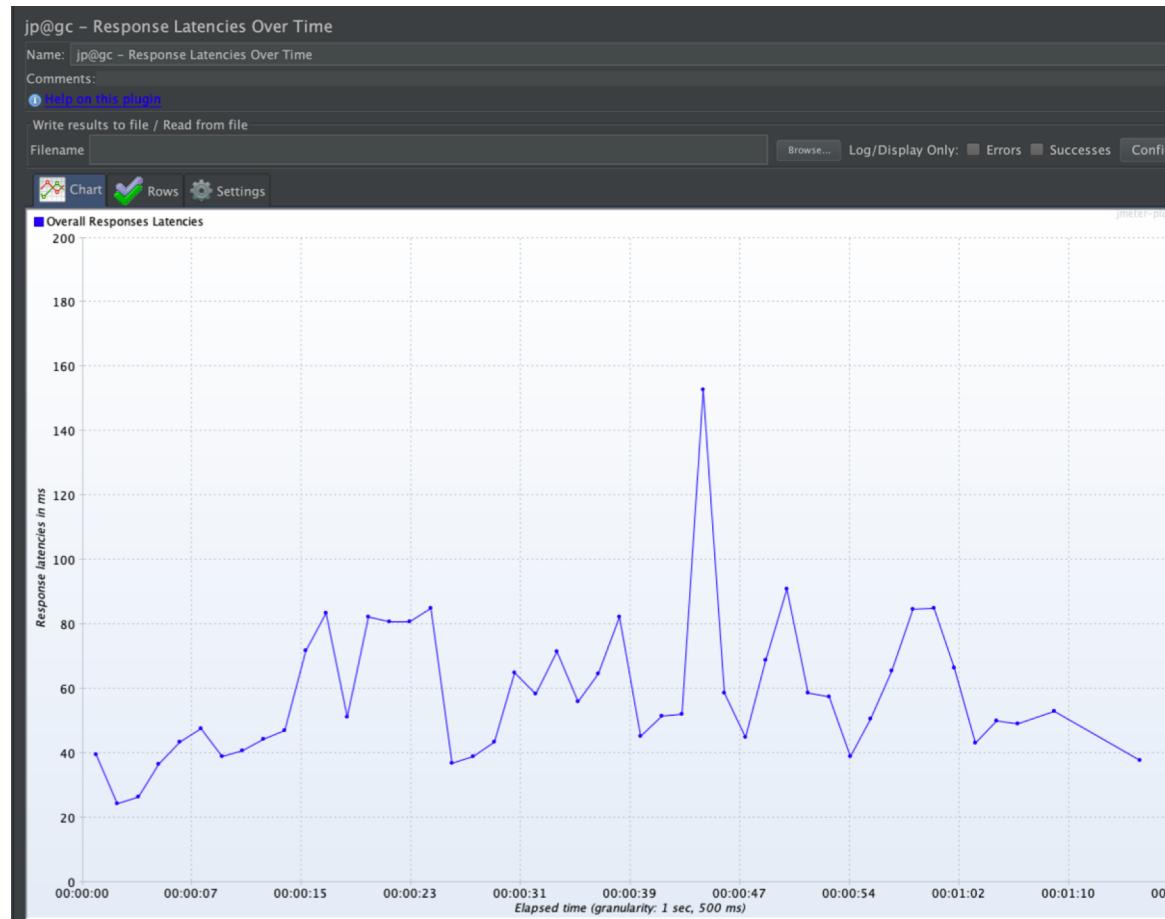
# Lanzamientos de Pruebas Módulo Ad hoc

28

## Prueba #2

### Resultado:

En este caso, los tiempos de latencia son bajos debido al ahorro computacional que proporciona el uso de cache, lo cual previene el colapso y acumulación de peticiones. Cuantitativamente el mayor pico que tenemos con cache es de 155 milisegundos, cuando sin cache se tuvo un pico de 9 segundos de latencia.



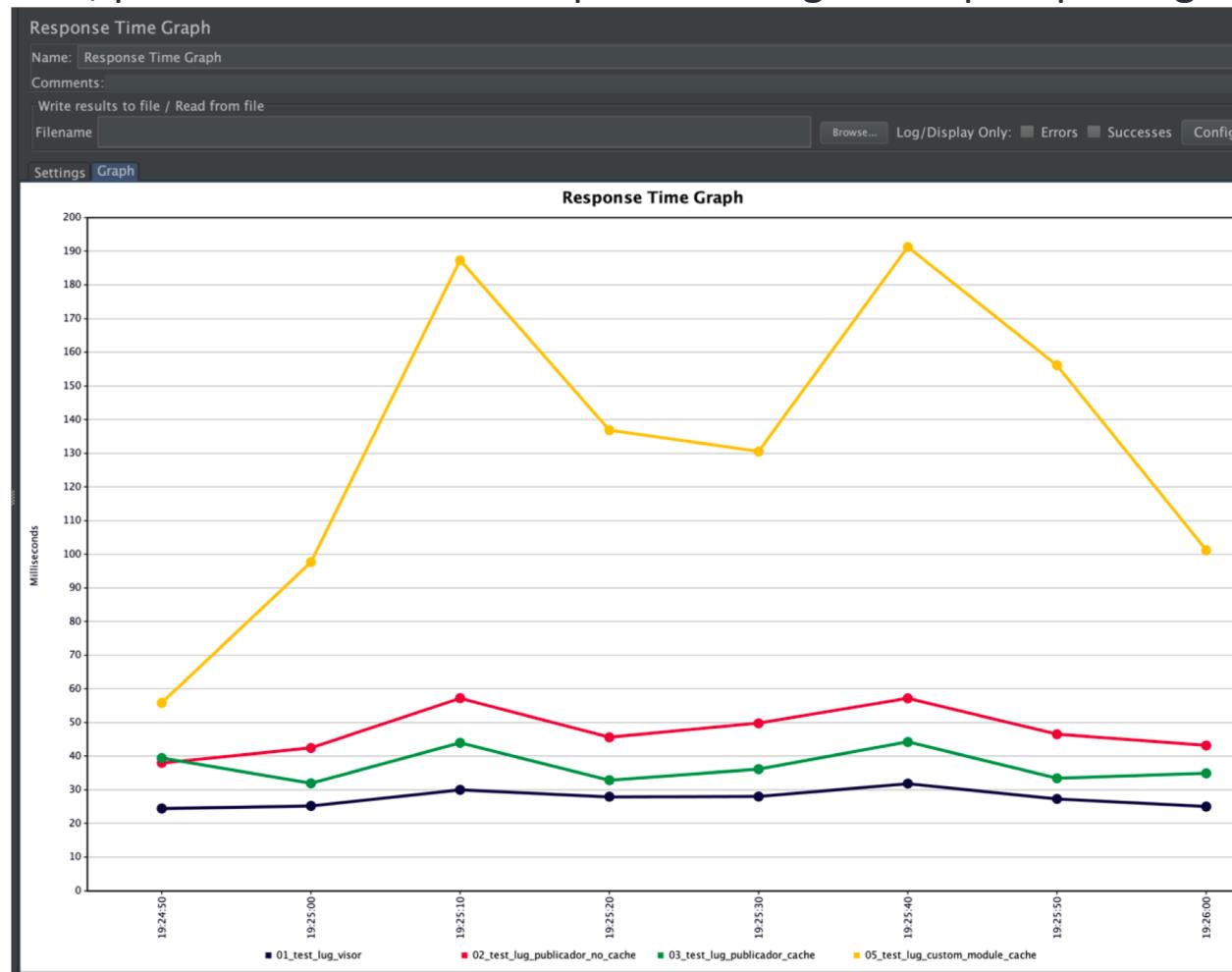
# Lanzamientos de Pruebas Módulo Ad hoc

29

## Prueba #2

### Resultado:

Al igual que ocurrió en el ejemplo anterior, los tiempos de respuesta del módulo Ad hoc siempre está por encima de los elementos OOTB, pero en este caso no se produce ningún colapso que degrade ninguna de las páginas.



# Lanzamientos de Pruebas Módulo Ad hoc

30

## Prueba #2

### Resultado:

Fijándonos en la influencia sobre la máquina, observamos que en cuando se utiliza la cache en el módulo ad hoc (actividad de las 19:25) la CPU está entorno al 50% del rendimiento, con picos del 67%, y no se produce apenas un aumento del número de hilos. Mientras que sin el uso de la cache (actividad de las 19:30) la CPU sufre mayores porcentajes de uso, llegando a picos del 100% y casi se triplica el número de hilos.



# Lanzamientos de Pruebas Módulo Ad hoc

## Prueba #2

### Resultado:

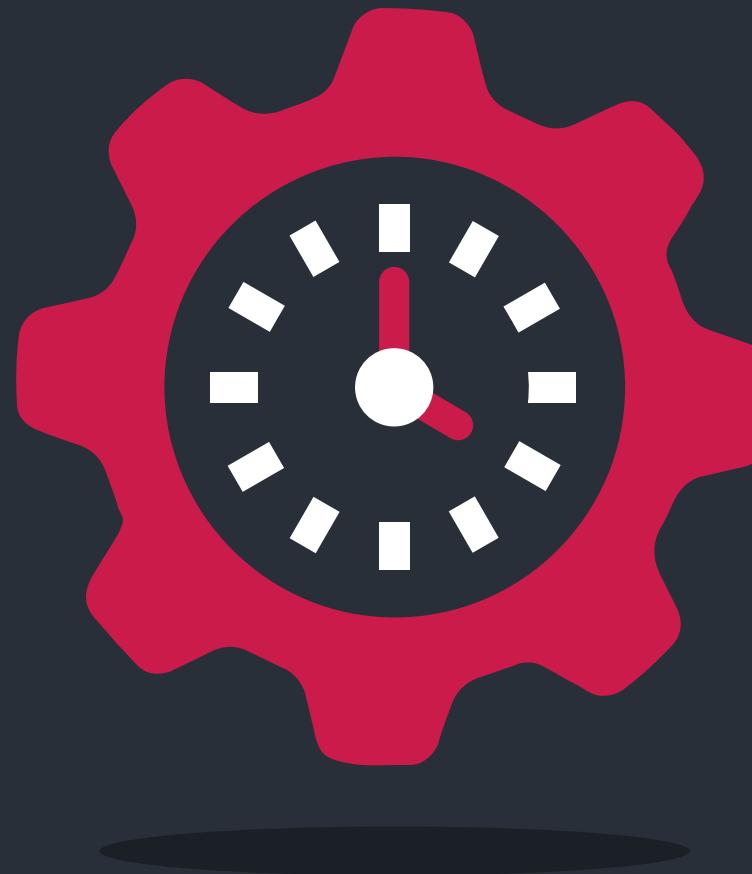
Por último, cabe mencionar el alto acierto generado en la cache usada para la prueba del módulo ad hoc con cache. Acertando en 1949 peticiones y fallando sólamente en 1 (la primera).

Attribute	Value
AssociatedCacheName	LUG_SPAIN
CacheHitPercentage	0.9994871794871795
CacheHits	1949
CacheMissPercentage	5.128205128205128E-4
CacheMisses	1
DiskStoreObjectCount	0
InMemoryHitPercentage	0.9994871794871795
InMemoryHits	0
InMemoryMisses	1949
MemoryStoreObjectCount	1
ObjectCount	1
OffHeapHitPercentage	0.0
OffHeapHits	0
OffHeapMisses	0
OffHeapStoreObjectCount	0
OnDiskHitPercentage	0.0
OnDiskHits	0
OnDiskMisses	0
WriterMaxQueueSize	0
WriterQueueLength	0

**NOTA:** En el caso de tener una degradación en el portal sería conveniente realizar un análisis de hilos con un thread dumps (Ver: [https://github.com/dmcisneros/lug\\_pruebas\\_carga/tree/master/thread\\_dumps](https://github.com/dmcisneros/lug_pruebas_carga/tree/master/thread_dumps) )

# ¡Lo vemos!

Lanzamientos de Pruebas Módulo Ad hoc – Prueba #2



# Performance Check List



## Portal-ext.properties

- com.liferay.portal.servlet.filters.\* (Desactivar Servlet filters no utilizados)
- session.tracker.memory.enabled=false (Deshabilitar session tracket si está activo)
- portlet.css.enabled=false (Ajustar la propiedad si no se va a utilizar personalización de apariencia desde portlets)
- locales.enabled= (Deshabilitar los que no se vayan a utilizar)
- dl.store.impl=com.liferay.portal.store.file.system.AdvancedFileSystemStore (Recomendada)
- direct.servlet.context.reload=false (En producción evitar la recarga de jsp en cada petición)
- counter.increment=2000 (Ajustar los indices de contadores)
- buffered.increment.standby.queue.threshold=60
- buffered.increment.standby.time.upper.limit=10000
- dl.file.rank.enabled=false
- dl.file.rank.check.interval=-1
- blogs.ping.google.enabled=false
- blogs.pingback.enabled=false
- blogs.trackback.enabled=false
- message.boards.pingback.enabled=false
- live.users.enabled=false
- session.tracker.memory.enabled=false
- session.tracker.persistence.enabled=false
- session.tracker.friendly.paths.enabled=false

# Check List



## Ajustar Session timeout

Reducir el tiempo de session provocará mantener menos objetos de memoria



## Ajustar server.xml

Ajustar server.xml con recomendaciones del fichero Deployment Checklist



## Ajustar ADT caché

En system settings ajustar la propiedad “resource modification check interval”, por defecto es 60ms



## Tunning de JDK

Tunning de JDK, GC, etc...



## Añadir elementos frontales para tratamiento de estáticos

Como podrian ser varnish, nginx, etc...



## Y más allá...

Cada Proyecto requerirá un tuning fino para optimizar el performance



# En resumen

Como desarrolladores debemos asegurarnos principalmente de que nuestro sistema funcionalmente sea lo que quiere el usuario final pero es igualmente importante asegurar la estabilidad y respuesta de nuestra arquitectura optimizando tiempos de respuesta, plan de contingencia ante caídas, asegurar la alta disponibilidad, etc...

# Preguntas?



David Vega Perea (@davidsrules7) - Minsait

Daniel Martínez Cisneros (@dmcisneros) - Everis