Motivación.
Resolver los problemas de incapacidad para absorber picos de crecimiento de consumo de recursos superiores al 1000 por ciento manteniendo estabilidad y calidad de servicio. Estos picos se deben a acciones puntuales como el envío de una push masiva, o la publicación en redes sociales, lo que produce un acceso masivo y de golpe de un número indeterminado de usuarios. Este efecto sobre los sistemas se ve multiplicado tanto por el mayor número de usuarios como por la manera en que son consumidos los servicios desde las apps desde el momento en que se inicia la aplicación.

Escenario.
Software Django funcionando en un entorno apache en contenedor docker servido en cluster kubernetes.

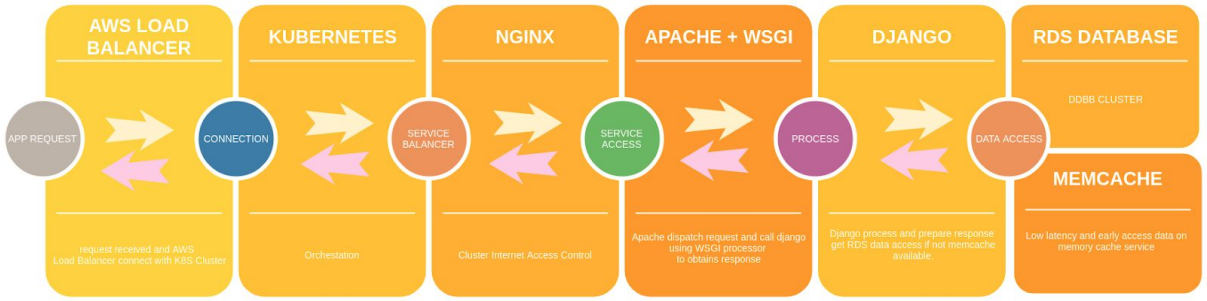
Problemas detectados.
Cuando hay cambios en el régimen normal de funcionamiento del servicio y llegan picos de usuarios, la memoria requerida por el software se desborda por encima de la ofrecida en los nodos del cluster, al sufrirse dicho pico el nodo llega a quedar en estado “Not Ready” por destinar sus recursos a tratar de dar el servicio y de ese modo el balanceo de carga se ve desvirtuado llevando toda la carga a otro nodo que también cae produciendo un efecto dominó que lleva a la caída absoluta del sistema, todo esto ocurre tan rápido que ningún mecanismo de auto escalado puede reaccionar a tiempo para absorber esta carga, obligandonos a sobre escalar los sistemas como medida preventiva. Implicando esto un enorme coste.

Factores.
Cualquier software funcionando sobre Apache con una configuración por defecto como norma general sin cuellos de botella en etapas posteriores del ciclo de vida de una petición es capaz de mantener un alto índice de respuesta y absorción de crecimiento de demanda.

Aunque el índice requerido es muy alto normalmente con técnicas de auto escalado puede gestionarse debido a que hay tiempo suficiente para que se instancian los sistemas y se vaya distribuyendo la carga. Sin embargo el software que manejamos, aunque ha sido mejorado con técnicas de caché sigue siendo muy intensivo en el consumo de base de datos, y esto hace que se encolen y saturen las peticiones en momentos oleadas de usuarios.

Por presupuesto, trabajamos con máquinas EC2 de AWS con limitados recursos de memoria y procesador que hay que gestionar muy bien, ya que cuando la memoria es desbordada los servidores tienden a colgarse.

Cuellos de botella, en cada una de las etapas del ciclo de vida de una respuesta puede producirse un cuello de botella, tanto en la gestión de la respuesta como en la velocidad de escalado en caso de saturación.



Reto.
Cómo es inviable rediseñar el software de backend en esta etapa se trata de alcanzar este enorme requerimiento con técnicas puramente de sistemas, para ello buscamos limitar y controlar el consumo de memoria de los contenedores de Django

Validaciones.

- los nodos de kubernetes se declaran “NotReady” cuando alcanzan el 100% de consumo de memoria ram necesaria para gestionar los procesos de control y orquestación.
- para que los servicios balanceen correctamente es necesario definir un valor de cpu request.
- para que los servicios realicen autoescala en pods (HPA) es necesario definir un valor de cpu request y crear el grupo de autoescalado.
- El micro servicio kafka-message tiene un memory leak que ha sido controlado mediante la configuración de una limitación de manera temporal.
- El servicio Dkron usado por RTS para agendar las acciones sufre saturación por el alto volumen de transacciones que maneja, demostrando no ser adecuado para el uso que le damos, (actualmente controlado por su consumidor RTS que borra los procesos terminados de forma activa) . (buscamos implantar o desarrollar una alternativa)

Hipótesis plateadas que han sido validadas o descartadas

- una mala configuración de Kubernetes puede ser la causa de la inestabilidad.
    - **descartado**. el sistema sufre los mismos efectos trabajando fuera de kubernetes, más adelante se muestran los resultados de las pruebas realizadas.
- rts, kafka, dkron u otros sistemas satelitales pueden ser la causa de la inestabilidad.
    - **descartado**. se desactivan estos sistemas y el comportamiento fue el mismo que con ellos en funcionamiento.
- modificando las configuraciones de Apache podemos cambiar el comportamiento bajo presión y ajustarlo a nuestras necesidades.
    - **confirmado**. podemos ajustando algunos parámetros contener la memoria, limitando así la capacidad de servicio de cada instancia pero garantizando que no haya explosiones en memoria, manteniendo un consumo estable y siendo la CPU la que sube o baja según el índice de demanda que sufra la instancia.

Implantación de Locust
Locust es un banco de pruebas, que mejora el que hasta ahora veníamos usando en Jmeter para estresar los sistemas, hemos realizado un empaquetado de este software para trabajar en docker, de manera que podemos desplegarlo en nuestros propios clusters y poder lanzar simulaciones realistas de avalanchas de peticiones.

Implantación de grafana y prometheus.
Sistemas que nos permiten trazar y monitorizar con precisión todo lo que sucede y han sido claves para determinar el origen de los problemas.

ejercicios realizados con resultado negativo:

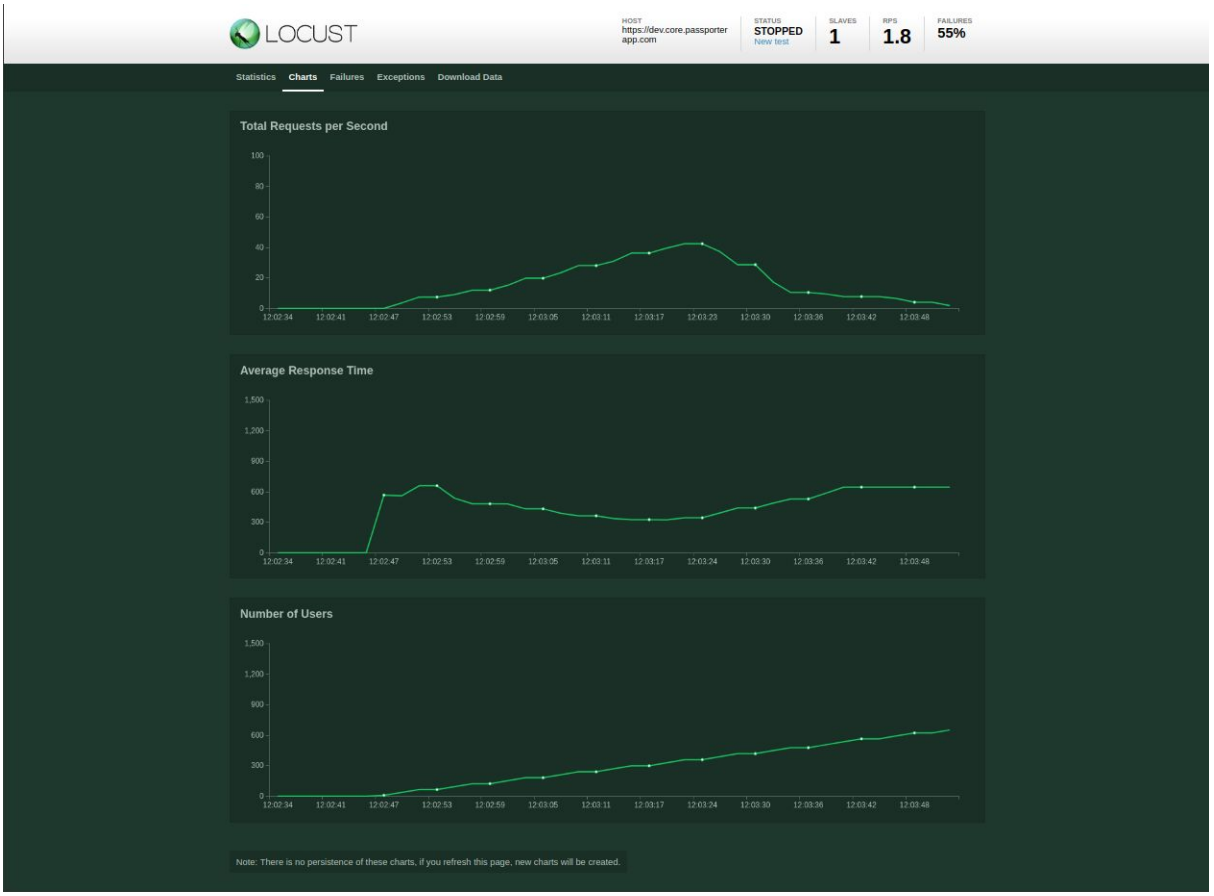
limitar la memoria que puede consumir el software desde kubernetes.
el sistema operativo del nodo mata el proceso que trata de desbordarse de memoria reiniciando, para evitar que este sature al sistema hospedador. Como consecuencia el servicio no se cae pero 1 de cada N peticiones se devuelve errónea debido a estos reinicios que aunque se efectúan en milésimas de segundo, llegan a producir este efecto adverso. Y en situaciones altas de demanda, el número N llega a ser muy bajo. 1 de cada 10 peticiones incluso llega a devolverse errónea.

probar diferentes sistemas de auto escalado sin gestionar el crecimiento explosivo de consumo de memoria al recibir picos wall o tsunami, sin crecimiento exponencial.
la manera en que se acelera en la demanda de recursos de memoria es tan intensa que la auto escala no puede dar solución a este problema, ya que no le da tiempo a instanciar nuevos sistemas antes de que los que están en funcionamiento se desborden.

montar un sistema externo a kubernetes con DJANGO funcionando en docker directamente en máquinas EC2 de AWS con auto escalado y a su vez externalización de los servicios redis y memcached necesarios para su funcionamiento.

en un principio el resultado fue adecuado, tras realizar las pruebas de carga pertinentes nos dispusimos a dirigir el trafico de producción a este sistema de manera transparente y todo funcionó correctamente, estuvimos sirviendo el tráfico desde este sistema durante aproximadamente 48 horas, y probamos la reacción a una push con el sistema preescalado y pudimos dar servicio.
Sin embargo tras dirigir el tráfico nuevamente a otro cluster, le hicimos una prueba de carga Tsunami usando locust con 1000 usuarios concurrentes y una cadencia de 5 segundos. El resultado fue caída absoluta del sistema en pocos segundos sin capacidad de auto escalar,

quedando colgados los sistemas y tardando del orden de 4 o 5 minutos el grupo de auto escalado en recuperarse instanciando nuevos sistemas.



como se puede ver en la prueba de locust, el crecimiento de 600 usuarios en 1 minuto ha sido suficiente para saturar el sistema dejándolo inoperativo, teniendo en cuenta que las nuevas instancias tardan en aparecer del orden de 5 minutos, es insostenible soportar un crecimiento de esta índole desde un estado de reposo.

Montar un cluster EKS autoescalado, con un nodeGroup destinado a Django sus servicios redis y memcache además de los sistemas de kube-system.
comprobamos la manera de escalar y desescalar de este cluster, y pudimos comprobar que lo hacía con mucha agilidad, en menos de un minuto podíamos disponer de una nueva instancia en el nodeGroup operativa en la que desplegar más pods de Django para balancear.
Tras comprobar y medir la capacidad dirigimos el tráfico real a este cluster durante 48 horas, pudiendo soportarlo, sin embargo validamos que al no disponer de nodo maestro por la propia arquitectura que propone AWS EKS, todo se gestiona a través de nodeGroups, de manera que tienes que configurar manualmente un nodeGroup diferente para cada sistema de responsabilidad, esto no encaja con nuestras necesidad de ahorro en sistemas. Además del coste fijo de 140 euros mes sobre el coste de servidores que supone el uso de la plataforma EKS.

Cuando estresamos el cluster una de las máquinas se saturó por desbordamiento de memoria de Django, y eso supuso que todo el sistema de auto escalado se bloquease al estar los componentes de kubernetes en el mismo nodeGroup.

ejercicios realizados con resultado positivo.

Configurar los nodeGroups en el cluster EKS para comprobar que los sistemas de gestión del cluster y los deployments desplegados no se afectan entre sí.

SOLUCIÓN ALCANZADA DE MANERA SATISFACTORIA.

Con todo lo aprendido y validado anteriormente, procedemos a configurar nuestro cluster KOPS, teniendo el trafico de producción dirigido a EKS podíamos hacer pruebas sin riesgo de downTime del servicio.

Aprovechando la versión de la imagen de DJANGO en la que hace unas semanas dejamos las variables de entorno que permiten configurar Apache de una manera sencilla sobre docker, hicimos diferentes pruebas de afinado llegando a esta configuración.

DEFAULT

ENV APACHE_MPM_START_SERVERS 5

ENV APACHE_MPM_MIN_SPARE_SERVERS 5

ENV APACHE_MPM_MAX_SPARE_SERVERS 10

ENV APACHE_MPM_MAX_REQUEST_WORKERS 256

DJANGO PASSPORTER CONFIG

APACHE_MPM_START_SERVERS: '2'

APACHE_MPM_MIN_SPARE_SERVERS: '2'

APACHE_MPM_MAX_SPARE_SERVERS: '5'

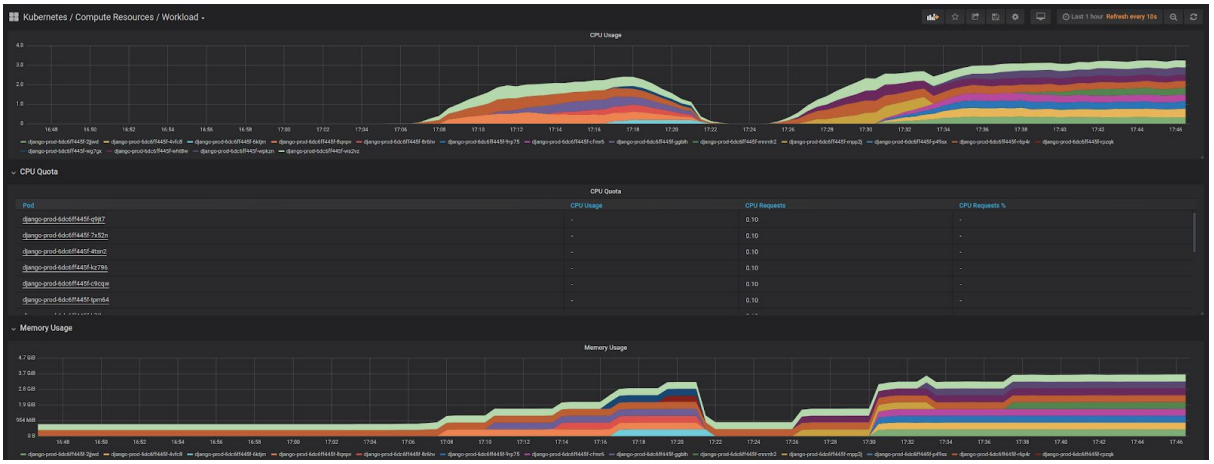
APACHE_MPM_MAX_REQUEST_WORKERS: '5'

APACHE_MPM_START_SERVERS - Número de workers al arrancar
APACHE_MPM_MIN_SPARE_SERVERS - Numero de workers mínimo
APACHE_MPM_MAX_SPARE_SERVERS - Número de workers máximo
APACHE_MPM_MAX_REQUEST_WORKERS - Número de peticiones máxima que acepta cada worker. Esto es lo que hace que la memoria y la cpu se dispare

Esta configuración hace que el consumo de ram sea constante pase lo que pase, sin disparos y sin riesgos de saturar las máquinas, con esto el mayor problema de inestabilidad está totalmente **RESUELTO**

A partir de aquí podemos proceder a configurar con garantías todos los sistemas de auto escalado del cluster ya que Django ahora si crece y decrece en el consumo de CPU manteniendo la ram totalmente estable. Siendo los indicadores de CPU los utilizados por los disparadores de auto escala.

Los resultados fueron inmejorables en el sentido de que podemos trabajar con máquinas medium en lugar de large que cuestan la mitad, ya que el consumo de un pod de DJANGO se acota a 500 megas frente a los 1,2 Gigas inestables pudiendo crecer hasta el infinito que teníamos antes y lo que provoca la saturación del cluster.



en la gráfica podemos ver como el cluster escala teniendo tan solo dos maquinas operativas en pocos minutos ante una prueba realmente intensa.
Hay dos ciclos de pruebas, el primero con el cluster en reposo, y el segundo con el cluster recién desescalado, teniendo en cuenta que las máquinas tras instanciarse se mantienen en reserva durante 10 minutos hasta que son desconectadas por el “cluster auto scaler”
En el segundo ciclo de pruebas, comprobamos que los servidores escalaron mucho más rápido en apenas segundos debido a la reserva del primer ciclo.

En la grafica de la prueba se ven también los dos ciclos comentados anteriormente.

En el primer ciclo comprobamos cómo pasa de 0 a 1000 usuarios en 1 minuto sin caida con una cadencia de 50 a 60 requests por segundo manteniendo la calidad de servicio.

En el segundo ciclo comprobamos un resultado similar, esta prueba es de una intensidad muy alta, equiparable a los tsunamis de usuarios llegados a través de pushes y campañas. Pero el banco de pruebas tocó techo.

El sistema Locust podemos montarlo para tener un nivel mucho más alto de intensidad en las pruebas, y tenemos previsto hacerlas sencillamente para medir con más precisión el índice de costes por usuario concurrente real que tenemos.

Ya que en este contexto podemos medirlo lo hacemos del siguiente modo. (sin contar los sistemas de desarrollo, y staging, tan solo los de producción y sin contar los sistemas satelitales. Tan solo el servicio directo a las apps.

1 usuario conectado realiza una media de 1 conexión cada 5 segundos al sistema y esta es la simulación establecida en Locust.

Pasamos a maquinas T3 en lugar de máquinas T2, debido a que estas ofrecen mejores prestaciones para nuestras necesidades en lo referente a consumo de CPU.