BALANCEO DE CARGAS DE SERVIDORES WEB USANDO NGINX Y ARTILLERY

Web server load balancing using Nginx and Artillery

Juan Sebastian Varona Parra[1], Jharling Jhoan Rodriguez Rodriguez[2], Ana Sofia Lopez Lopez[3], Juan Sebastián Losada Sanchez[4], Julián David Velásquez Patiño[5]

Afiliación institucional de los autores: Universidad Autónoma de Occidente
Facultad de Ingeniería, Departamento de Informática
Cali, Colombia

juan.varona@uao.edu.co[1] - jharling.rodriguez@uao.edu.co[2] - ana_sof.lopez@uao-edu.co[3] - juan.losada@uao.edu.co[4] - julian_dav.velasquez@uao.edu.co[5]

Resumen – En este proyecto se implementa un clúster de servidores web con balanceo de carga, configurando el balanceador como frontend para redirigir las peticiones a los servidores backend. Se desarrolla y ejecuta un plan de prueba de carga para simular múltiples peticiones, observando el número máximo de usuarios concurrentes en el sistema.

Finalmente, se analizan los resultados para identificar el rendimiento y la estabilidad del clúster, garantizando la capacidad para manejar eficientemente la carga esperada.

Abstract - In this project, a cluster of web servers with load balancing is implemented, configuring the balancer as a frontend to redirect requests to the backend servers. A load test plan is developed and executed to simulate multiple requests, observing the maximum number of concurrent users on the system.

Finally, the results are analyzed to identify the performance and stability of the cluster, ensuring the ability to efficiently handle the expected load.

I. INTRODUCCIÓN

Hay una cantidad significativa de entornos web donde las personas deben acceder eficazmente cuando lo soliciten. De esta forma la implementación de un balanceador de cargas resulta necesario para que el factor de disponibilidad este en juego para responder a todas las peticiones solicitadas.

En este proyecto se explicará de manera detallada una forma de implementar un balanceador de carga usando Nginx, realizando las pruebas desde Artillery, con las cuales se busca verificar el número de usuarios concurrentes que la aplicación soportará y que carga hará que la aplicación deje de funcionar.

II. OBJETIVOS

A. General

Implementar un clúster de servidores web con balanceo de carga, asegurando su capacidad de gestionar múltiples peticiones simultáneas de manera eficiente, verificando el rendimiento y la estabilidad del sistema mediante pruebas de carga.

B. Específicos

- Configurar el clúster de servidores web con balanceo de carga, asegurando que las peticiones sean redirigidas adecuadamente a los servidores backend.
- Ejecutar un plan de pruebas de carga para simular múltiples peticiones simultáneas, determinando el número máximo de usuarios concurrentes que la aplicación puede soportar
- Analizar los resultados de las pruebas de carga, visualizando el rendimiento y la estabilidad del clúster de servidores web.

III. CONTEXTO

El conflicto que se está presentando está relacionado con la gestión eficaz de peticiones que se realizan a un solo servidor en un entorno de alto tráfico. En este contexto, la cantidad de peticiones en tiempo real genera sobrecarga en el servidor, dificulta el manejo del volumen de peticiones y resulta en tiempos de respuesta lentos y fallos en el sistema. Además, se compromete la

disponibilidad al no tener un sistema de respaldo adecuado, cualquier fallo o baja del servidor desembocaría en la interrupción completa del servicio, afectando a todos los cliente o usuarios que dependen de él.
De igual forma la escalabilidad del sistema se ve limitada. A medida que el número de usuarios crece, el sistema actual no ofrece la flexibilidad necesaria para manejar la carga adicional de manera efectiva. Esto impide un crecimiento sostenido y puede llevar a una degradación del rendimiento del servicio.

IV. ALTERNATIVAS DE SOLUCIÓN

Considerando la problemática previamente descrita, es crucial destacar diversas soluciones para equilibrar eficientemente las cargas en los servidores web. En consecuencia, se presentan las siguientes alternativas:

1. **Balanceador de carga usando el módulo de apache mod_proxy_balancer**

   En el caso de Apache, el módulo mod_proxy_balancer permite al servidor web actuar como un proxy inverso reenviando las solicitudes a otro servidor y devolviendo la respuesta al cliente. Esto es muy útil cuando se tienen varios servidores web y se busca distribuir la carga entre ellos.

2. **Balanceador de carga usando NGINX**

   NGINX es un servidor web/proxy inverso ligero de alto rendimiento y un proxy para protocolos de correo electrónico. Es conocido por su alta performance, estabilidad, conjunto de características, configuración simple y bajo consumo de recursos. Este también puede actuar como un balanceador de carga que distribuye las solicitudes entrantes a múltiples servidores. Esto se utiliza para aumentar la disponibilidad, confiabilidad y escalabilidad de las aplicaciones.
   También permite agregar muchos servidores cuando aumenta el tráfico, lo que lo hace una solución interesante.

3. **Balanceador de carga usando AWS Elastic Load Balancing**

   Elastic Load Balancing es un servicio de Amazon Web Services (AWS) que distribuye automáticamente el tráfico de aplicaciones entrantes entre varios destinos, como instancias EC2, contenedores y direcciones IP, en una o varias zonas de disponibilidad. Este "equilibrador de carga inteligente" monitorea constantemente el estado de los destinos registrados y solo envía tráfico a aquellos que están en buen estado, ajustando automáticamente su capacidad para manejar cambios en el tráfico entrante. De esta manera, permite lograr mayores niveles de tolerancia a fallos en las aplicaciones y proporciona la cantidad necesaria de capacidad de equilibrio de carga para distribuir eficientemente el tráfico.

Al analizar la tabla comparativa (*ver anexo 1*) de las tres alternativas de solución para el balanceo de carga de servidores web, se observa una clara diferenciación en términos de rendimiento, configuración, seguridad, costos e integración. NGINX destaca por su alta eficiencia y baja latencia, ofreciendo flexibilidad y características de seguridad robustas, sin costos adicionales debido a su naturaleza de código abierto. Sin embargo, su complejidad y la necesidad de conocimientos avanzados en Linux/Unix pueden representar una barrera para usuarios novatos. Por otro lado, mod_proxy_balancer, integrado en Apache, presenta una configuración sencilla y sin costos adicionales, pero su rendimiento y escalabilidad son menos eficientes en comparación con NGINX y AWS Elastic Load Balancer. AWS Elastic Load Balancer, optimizado para alta disponibilidad y escalabilidad automática, simplifica la gestión mediante la consola de AWS y ofrece una integración profunda con otros servicios de AWS, aunque su modelo de costos basados en uso puede resultar más caro a largo plazo. Además, depende completamente de la infraestructura de AWS y puede ser complejo para configuraciones avanzadas.

Teniendo en cuenta lo anterior, NGINX es ideal para quienes buscan rendimiento y flexibilidad sin costos recurrentes, mod_proxy_balancer es adecuado para aplicaciones basadas en Apache con necesidades medianas, y AWS Elastic Load Balancer es la mejor opción para quienes requieren una solución gestionada y altamente escalable en la nube, dispuestos a asumir los costos asociados.

V. DISEÑO DE LA SOLUCIÓN

A. Balanceador de cargas

Este es un dispositivo hardware o software que distribuye las peticiones de los clientes a un conjunto de servidores que atienden un servidor. Este usa algoritmos

y políticas predefinidas para asignar aquellas solicitudes y evitar la sobrecarga de los servidores. Debido a esto, el objetivo del balanceador de carga maximiza la tasa de datos y minimiza los tiempos de respuesta.

B. Arquitectura del Sistema

La solución propuesta incluye un balanceador de carga configurado con NGINX y tres servidores backend, todos ellos gestionados en un entorno de máquinas virtuales usando Vagrant. Para las pruebas de rendimiento y carga, se utilizará Artillery.

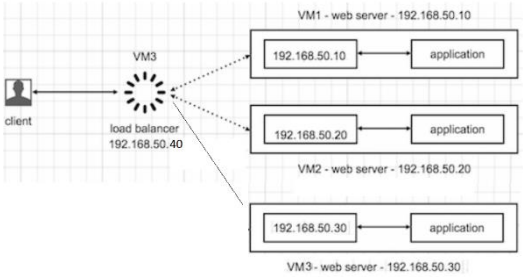


Figura 1. Arquitectura funcionamiento de un balanceador de carga

C. Servidores Web

Son programas o sistemas software que responden a solicitudes de clientes, generalmente navegadores web enviando recursos como páginas web. Estos recursos se alojan en el servidor y se envían al cliente cuando se realiza una solicitud mediante el protocolo HTTP. Estos son fundamentales para la infraestructura de internet y permiten que los usuarios accedan y visualicen en www.

Descripción de las Herramientas

D. Vagrant

Vagrant es una herramienta de software para la creación y configuración de entornos de desarrollo virtualizados. Utiliza configuraciones automatizadas para gestionar máquinas virtuales, proporcionando un entorno de desarrollo consistente y portátil.

Se hace uso de esta herramienta para crear y gestionar máquinas virtuales Ubuntu 20.04, facilitando la replicación del entorno de desarrollo y asegurando consistencia en las configuraciones.

E. NGINX:

NGINX es un servidor web y proxy inverso de alto rendimiento que incluye capacidades de balanceo de carga y proxy para HTTP, HTTPS, SMTP, POP3 e IMAP.

Se realiza la configuración con su módulo upstream para balancear la carga entre los tres servidores backend, mejorando la distribución de peticiones y optimizando el uso de recursos del servidor.

F. Artillery:

Artillery es una herramienta de pruebas de carga y rendimiento que permite simular tráfico de usuarios y medir el rendimiento de aplicaciones web y servicios.

Es utilizada para generar cargas de prueba que evalúen la capacidad del balanceador de carga NGINX y los servidores backend bajo condiciones de alta demanda.

VI. IMPLEMENTACIÓN

Configuración del Vagrantfile

Se especifican las máquinas necesarias con su respectiva configuración de red.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|

    if Vagrant.has_plugin? "vagrant-vbguest"
        config.vbguest.no_install = true
        config.vbguest.auto_update = false
        config.vbguest.no_remote = true
    end
    config.vm.define :servidor1 do |servidor1|
        servidor1.vm.box = "bento/ubuntu-22.04"
        servidor1.vm.network :private_network, ip: "192.168.50.10"
        servidor1.vm.hostname = "servidor1"
        servidor1.vm.boot_timeout = 600
    end
    config.vm.define :servidor2 do |servidor2|
        servidor2.vm.box = "bento/ubuntu-22.04"
        servidor2.vm.network :private_network, ip: "192.168.50.20"
        servidor2.vm.hostname = "servidor2"
        servidor2.vm.boot_timeout = 600
    end
    config.vm.define :servidor3 do |servidor3|
        servidor3.vm.box = "bento/ubuntu-22.04"
        servidor3.vm.network :private_network, ip: "192.168.50.30"
        servidor3.vm.hostname = "servidor3"
        servidor3.vm.boot_timeout = 600
    end
    config.vm.define :balanceador do |balanceador|
        balanceador.vm.box = "bento/ubuntu-22.04"
        balanceador.vm.network :private_network, ip: "192.168.50.40"
        balanceador.vm.hostname = "balanceador"
        balanceador.vm.boot_timeout = 600
    end
end
```

Figura 2. Configuración Vagrantfile

Configuración de los servidores

Se procede a la instalación de NGINX en cada máquina virtual

```
#ingresamos como super usuario
sudo -i
# actualizamos paquetes
apt-get update
# instalacion de Nginx
apt-get install nginx -y
```

Figura 3. Instalación de NGINX

Posteriormente, se modifica el archivo index.html ubicado en el directorio /var/www/html de los servidores 1, 2 y 3

```
<!DOCTYPE html>
<html>
<head>
<title>Servidor 1</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>servidor 1</h1>

 <p>Este es el servidor 1.</p

</body>
</html>
```

Figura 4. Configuración Archivo index.html

El proceso es el mismo para todos servidores cambiando únicamente el cuerpo del archivo.

Configuración del balanceador

```
#ingresamos como super usuario
sudo -i
# actualizamos paquetes
apt-get update
# instalacion de Nginx
apt-get install nginx -y
# se elimina el archivo de configuración predeterminado de Nginx
rm -rf /etc/nginx/sites-enabled/default
# se crea un nuevo archivo de configuración del balanceador de carga
vim /etc/nginx/conf.d/load-balancing.conf
```

Figura 5. Configuración Balanceador

Se añaden las siguientes líneas:

```
upstream backend {
       server 192.168.50.10;
       server 192.168.50.20;
       server 192.168.50.30;
                           }

   server {
           listen      80;

           location / {
               proxy_redirect      off;
               proxy_set_header    X-Real-IP $remote_addr;
               proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
               proxy_set_header    Host $http_host;
               proxy_pass http://backend;
           }
   }
```

Figura 6. Configuración Archivo load-balancing.conf

Finalmente, se reinicia el servicio NGINX para aplicar los cambios en todas las máquinas.

Al finalizar se debe observar algo lo siguiente:

Figura 7. Respuesta Balanceador Servidor 1

Figura 8. Respuesta Balanceador Servidor 2

Figura 9. Respuesta Balanceador Servidor 3

Se observa que la dirección es la misma para todos, lo que indica que la petición del cliente está dirigida hacia el balanceador de carga, y es este quien se encarga de redirigir la petición a cada uno de los servidores del clúster.

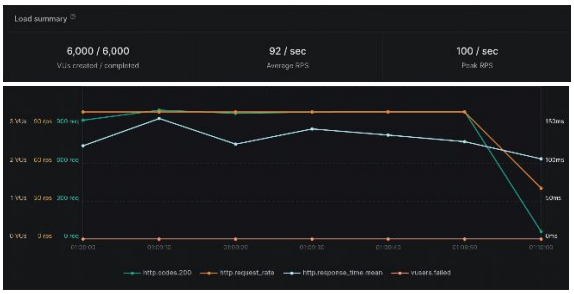
VII. PRUEBAS

Una vez realizada la configuración del clúster de servidores web con el balanceador de carga, se realizaron una serie de pruebas mediante el software de Artillery, donde se lograron evidenciar a través de las diferentes gráficas entregadas cómo se comporta la implementación del balanceador de carga en diferentes escenarios.

En un principio, se realizaron pruebas de carga con dos servidores web, con 100, 160 y 180 usuarios, en 60 segundos, y en cada segundo aumentaron los usuarios.

A continuación, se presentan los resultados obtenidos con la prueba realizada durante 60 segundos, incrementando durante cada segundo en 100 el número de usuarios.

Prueba de 100rps con 2 servidores:



Gráfica 1. Prueba 100 rps con 2 servidores

Como se puede observar el servidor logró completar la prueba sin ningún inconveniente ya que todas las solicitudes fueron completadas exitosamente. El tiempo de respuesta también se mantuvo estable durante la prueba. También podemos observar que los status_codes 200 se acercan bastante al request_rate que se manejó por lo que podemos concluir que el servidor cumplió con los requisitos de carga de la prueba.

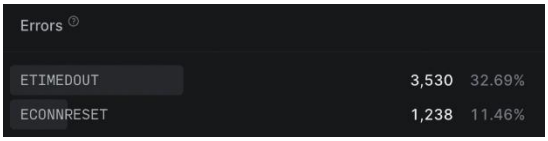
Prueba de 160rps con 2 servidores:

Gráfica 2. Prueba 160 rps con 2 servidores

Para esta prueba podemos ver que todas las solicitudes se completaron sin errores. En la prueba, el tiempo de respuesta también se mantuvo estable, pero cerca del final, subiendo exactamente en los últimos 20 segundos. También podemos observar que en el inicio de la prueba los status_codes 200 empezaron un bastante más abajo del request_rate hasta que se normalizó y luego volvió a bajar acercándonos al final.

Prueba de 180 con 2 servidores:

Gráfica 3. Prueba 180 rps con 2 servidores



Gráfica 4. Errores Prueba 180 rps con 2 servidores

Como podemos ver, para esta prueba el porcentaje de error fue del 44.15%. El request_rate tampoco se pudo mantener estable como en las pruebas anteriores y fue bastante disparejo durante la ejecución de la prueba. Además de ser el más bajo en promedio siendo de tan solo 47 request por segundo. Los status_codes 200 se mantuvieron muy por debajo del request_rate en la mayoría de las peticiones. El tiempo de respuesta tampoco fue estable en ningún momento de la prueba y presentó bastantes picos con pendientes considerables. Por último, encontramos que la linea de vusers.failed que se mantuvo en 0 constante con las 2 pruebas anteriores fue bastante alta por lo general.

PRUEBAS CON 3 SERVIDORES

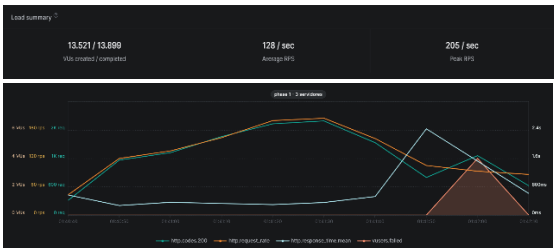
Luego de esta primera etapa de pruebas, se prosiguió a realizar una segunda etapa con tres servidores, con 150, 210 y 220 usuarios por segundo en cada prueba respectivamente.

Prueba de 150 rps con 3 servidores:

Gráfica 5. Prueba 150 rps con 3 servidores

Esta primera prueba con 3 servidores se realizó con el objetivo de comparar el rendimiento de los 3 servidores respecto a las pruebas anteriores con 2 servidores. Se puede observar un mejor desempeño en la disminución del tiempo promedio de respuesta frente a las solicitudes, también una mayor cantidad de los status_codes 200 es mayor que en la prueba más similar a esta de la primera etapa (160 rps), ya que solamente se empiezan en un nivel más bajo del request_rate, pero a medida que pasa el tiempo se normaliza y no vuelve a caer.

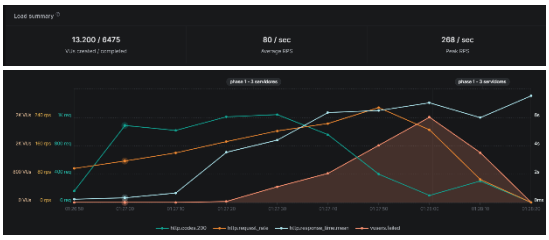
Prueba de 210 rps con 3 servidores:



Gráfica 6. Prueba 210 rps con 3 servidores

Esta prueba se realizó durante un total de 90 segundos, se puede observar cómo durante los primeros 50 segundos el tiempo de respuesta promedio fue bajo, además de no haber ningún error en la creación de vusers a pesar de que los codes 200 estaban ligeramente bajos respecto al rate de peticiones. Luego de esos primeros 50 segundos, se puede observar cómo empieza a disminuir el rate de peticiones junto a los status_code 200. Simultáneamente empezó a aumentar el tiempo medio de respuesta de forma acelerada alcanzando un pico a los 70 segundos de ejecución con un tiempo de 2434.3 ms. Posterior a este pico podemos ver que hubo un fallo en la creación de 4 VUs, lo cual representa un porcentaje de error del 4.30% respecto al número de peticiones realizadas.

Prueba con 220 rps con 3 servidores:



Gráfica 7. Prueba 220 rps con 3 servidores

En esta última prueba, ejecutada igualmente por 90 segundos, se puede ver como durante los primeros 30 segundos de ejecución hay un gran desempeño por parte de los servidores, el tiempo promedio de respuesta es bajo a medida que van aumentando las peticiones. De ahí en adelante se puede ver como siguen aumentando las peticiones, pero el tiempo promedio de respuesta también aumenta, mientras que disminuye el número de status_code 200 respecto al número de solicitudes y aumenta el porcentaje de error llegando a fallar más del 100% de VUs creados hasta ese momento (el pico fue de 2047 VUs a los 70 segundos). En cuanto al tiempo promedio de respuesta, el más alto llegó a los 7525 ms al final de la prueba.

VIII. CONCLUSIONES

Para ambas pruebas el procedimiento a llevar a cabo fue probar 3 escenarios donde se pueda evidenciar el comportamiento de los servidores bajo una carga leve fácil de manejar para ver el funcionamiento ideal. Luego se realizaron pruebas en un escenario donde los servidores contaban con una carga que empezaba a estresar moderadamente los servidores para entender después de que punto se empezaba a bajar el rendimiento. Por último, se trabajó en un escenario de alto estrés donde los servidores apenas podían terminar la prueba sin romperse.

Se logra observar que el balance loader manejando 3 servidores tuvo una mayor capacidad de manejar peticiones que el de 2 servidores como se esperaba. Al contar con mayores recursos para manejar las cargas.

En ambas pruebas vimos que los servidores suelen tener inicios algo más lentos y suelen normalizar la cantidad de peticiones que pueden manejar al cabo de unos 10 segundos. Y dependiendo del nivel de estrés también tienden a decrecer este número acercándonos al final de la prueba. También notamos que en situaciones de estrés moderado/bajo el tiempo de respuesta tiende a ser relativamente lineal en comparación con situaciones de estrés alto donde estos tiempos pueden fluctuar mucho y presentar bastantes picos.

También cabe destacar que en situaciones de cargas muy altas los servidores bajan considerablemente el response_rate que pueden manejar. Dejando de ser este lineal como en situaciones normales. Logrando completar muchas menos peticiones que cuando se presentan cargas moderadas y aumentando considerablemente el índice de error a su vez.

Teniendo en cuenta lo anterior, se puede concluir con respecto a las pruebas realizadas y los escenarios ideales para que nuestros servidores sean lo más eficientes posible, debemos contar con servidores capaces de manejar la carga de peticiones por segundo con un 60%-70% de la capacidad máxima de estos. Esto porque vimos que los servidores tienden a comportarse mucho mejor en escenarios de estrés bajo/moderado manteniendo sus estadísticas lo más lineales posibles. Este porcentaje también se debe a que lo ideal es dejar un margen por si se presentan picos inesperados el servidor cuente con los recursos necesarios para poder manejarlos de forma eficiente y con la menor cantidad de errores.

IX. REFERENCIAS

Artillery Software, Inc. (2023). *Artillery Docs*. Obtenido de Artillery Docs:

https://www.artillery.io/docs/reference/test-script

AWS. (2024). *Amazon Web Services*. Obtenido de Amazon Web Services: https://docs.aws.amazon.com/es_es/elasticloadbalancing/latest/userguide/what-is-load-balancing.html

Díaz-Heredero, R. A. (22 de 2 de 2018). *Adictos al Trabajo - Autentia*. Obtenido de Adictos al Trabajo - Autentia: https://adictosaltrabajo.com/2018/02/22/tests-de-rendimiento-con-artillery/

HAProxy. (2024). *HAProxy Comunity Edition*. Obtenido de HAProxy Comunity Edition: https://www.haproxy.org/

IONOS. (3 de 11 de 2023). *Digital Guide IONOS*. Obtenido de Digital Guide IONOS: https://www.ionos.es/digitalguide/servidores/configuracion/configurar-apache-como-proxy-inverso/

Traefik. (2024). *Traefik Labs*. Obtenido de Traefik Labs: https://doc.traefik.io/traefik/

ANEXOS

Anexo 1:

Tabla Comparativa			
Característica	**Nginx**	**mod_proxy_balancer**	**AWS Elastic Load Balancer**
Ventajas			
Rendimiento	Alta eficiencia y baja latencia	Suficiente para aplicaciones de tamaño medio	Optimizado para alta disponibilidad y escalabilidad automática
Configuración	Flexible y altamente configurable	Integrado en Apache, fácil de configurar	Fácil de usar y gestionar desde la consola AWS
Seguridad	Buenas características de seguridad	Compatible con la seguridad de Apache	Integrado con AWS WAF y otros servicios de seguridad AWS
Costos	Open source, sin costos adicionales	Incluido con Apache, sin costos adicionales	Costos basados en uso, puede ser más costoso a largo plazo
Integración	Compatible con muchas aplicaciones y servicios	Fácil integración con aplicaciones basadas en Apache	Integración profunda con otros servicios AWS
Desventajas			
Complejidad	Puede ser complejo para usuarios novatos	Limitado en funcionalidad comparado con Nginx	Puede ser complejo para configuraciones avanzadas
Escalabilidad	Requiere configuración manual para escalar	Menos eficiente en escalabilidad	Escalabilidad automática pero dependiente de la infraestructura AWS
Soporte	Comunidad activa, pero soporte comercial limitado	Principalmente soporte de la comunidad Apache	Soporte profesional a través de AWS
Dependencia	Requiere conocimientos en Linux/Unix para configuraciones avanzadas	Depende de Apache, no es un producto independiente	Dependencia de la infraestructura de AWS
Mantenimiento	Requiere mantenimiento y actualizaciones manuales	Requiere mantenimiento junto con Apache	Mantenimiento y actualizaciones gestionadas por AWS