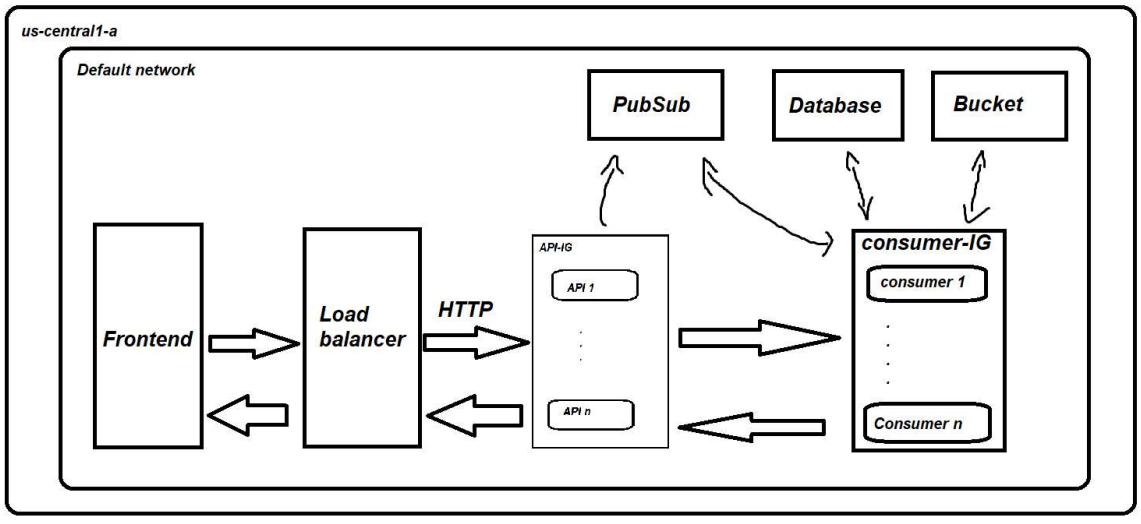
Taller 3 – Sistema Conversión Cloud

Gina Eveling Posada, Martin Daniel Rincón, Juan Camilo Muñoz, Felipe Serrano
MINE semestre 202410
Universidad de los Andes, Bogotá, Colombia
{g.posadas, md.rincon, jc.munozc12, ff.serrano42}@uniandes.edu.co
Fecha de presentación: abril 09 de 2024
github, video sustentacion

1) Arquitectura de referencia

La arquitectura se ilustra en la siguiente imagen. El frontend no tiene mayor cambio respecto a la entrega anterior, solo que ahora apunta a un balanceador de carga que se encarga de distribuir el tráfico http entre las instancias del grupo de API. Tanto el API como el consumer se implementan en contenedores docker que corren en máquinas virtuales; las máquinas de API y consumer se implementan en un "instance group", el cual se encarga de crear o eliminar instancias de acuerdo con las exigencias de recursos computacionales de la aplicación. El manejador de colas que antes se implementaba con rabbit en un contenedor que corría en la misma VM que el consumer, ahora se implementa con el servicio PubSub de GCP; el API escribe en las peticiones que luego son leídas por el consumer, quien se encarga de hacer la conversión y entregar archivos solicitados. La base de datos se implementa con el servicio de postgresql de GCP, su función es almacenar las direcciones en el bucket de los archivos. El bucket almacena los archivos y es accesible desde el consumer para borrar o insertar información.



Características y limitaciones de la infraestructura en producción:

- Proveedor Nube GPC cada estudiante cuenta con 1 cuenta con crédito de 50 usd para consumo en servicios de GCP.
- Redes e infraestructura con limitaciones de tiempo y restricciones que establece el proveedor GCP:
    - Reinicio de ips públicas de manera automática

- Bloque de puertos que pueda el proveedor de servicios de nube implementar
- Configuración de puertos de salida y entrada en cada máquina virtual

- Contenedores Docker: La aplicación actualmente está Dockerizado para todo su funcionamiento, es decir que se cuentan un contenedor por cada componente:
  - 1 contenedor Docker para el frontend: Streamlit
  - 1 contenedor Docker para el backend: Fast api
  - 1 contenedor Docker para el Worker consumidor: Python
- Servicios de GCP usados: SQL,PubSub,LoadBalancer,GCP storage.

2) Paso a paso Despliegue en una nube pública (GCP-AWS):

a) Descargar las siguientes imágenes desde docker hub

docker pull ffserrano42/fastapi:latest--> imagen de fastapi en máquina del api

docker pull ffserrano42/t1_streamlit:latest--> Imagen del streamlit en máquina del front

docker pull ffserrano42/consumer: latest--> Consumidor de la cola en máquina del worker

b) Comandos de docker útiles

docker rm [id del contenedor] --> elimina el contenedor

docker rmi [id de la imagen] --> elimina la imagen

-v /home/[nombreusuariogcp]/remote_folder:/remote_folder à hace la adición de una ruta local a docker para que puedan hacer llamados los programas a una ruta en específico.

c) Pasos para levantar la app por primera vez

1. Crear una base de datos postgres dentro de la red default en el servicio de SQL de GCP.
2. Crear topico y suscriptor en PubSub.
3. Crear un bucket de cloud storage.
4. Crear 3 máquinas virtuales y y descargar contenedores docker respectivos: máquina de API, máquina de Frontend, y máquina de worker.
5. Iniciar los contenedores con la bandera de reinicio automático y teniendo en cuenta los parámetros de los objetos creados en los pasos 1,2, y 3.
6. Parar las máquinas de worker y api para luego crear una imagen de cada una, un template, y por último un grupo de instancias.
7. Crear un balanceador de carga http con el "public facing" habilitado. En la configuración de frontend se pone el puerto del API (5001). En la configuración de backend seleccionamos el instance group del backend.

3) Limitaciones de la solución

Considerando lo representado, hay que considerar la escalabilidad vertical, ya que los recursos de CPU y Disco I/O no aumentan proporcionalmente con el aumento de la carga, lo que degrada el rendimiento con grupos de usuarios más grandes (30 y 40 usuarios). Existen cuellos de botella

significativos cuando se incrementa el número de usuarios y la carga de trabajo simultáneos, por lo que, los tiempos de respuesta y latencia aumentan dramáticamente con 30 y 40 usuarios, indicando que la infraestructura actual no puede manejar eficientemente cargas altas.

El escalado automático es inefectivo, ya no se activa a tiempo o no se escalan suficientes recursos para manejar los picos de carga, lo cual se muestra en la falta de uso del WORKER_AUTOSCAL para el grupo de 10 usuarios y el despliegue tardío de máquinas adicionales para los grupos de 20, 30 y 40 usuarios.

Es fundamental, revisar la configuración del escalado automático y la capacidad de las máquinas para garantizar que los recursos se escalen de manera proactiva y eficiente en respuesta a los picos de carga. Además, los recursos de CPU por contenedor se ven altamente utilizados en cargas mayores, lo que podría indicar que la aplicación podría beneficiarse de una mejor distribución de cargas.

4) Conclusiones

1) Por cuestiones económicas es necesario poner un límite superior a la cantidad de recursos que se pueden escalar, esto hace que puedan darse casos en que el auto escalado no pueda soportar la carga de los usuarios. Es importante considerar el número de máquinas y la capacidad de estas a la hora de calibrar los grupos de auto escalado.
2) Las máquinas virtuales toman un tiempo en estar listas para funcionar, por lo que si se el porcentaje de uso máximo antes de activar otra maquina es muy alto, o la carga crece muy rápido, puede haber franjas de tiempo en las que la aplicación no responda con la rapidez esperada por los usuarios.
3) GCP cuenta con varios servicios que nos pueden facilitar el desarrollo al no tener que implementar nosotros mismos; por ejemplo, un administrador de bases de datos relacionales, un sistema de almacenamiento, o un sistema de colas y suscripciones.