**Problemática**

Cada día a las 2am tengo que hacer un proceso bash, ¿en qué consiste ese bash?,a través de una petición POST a un endpoint de un sistema tercerizado llamado "CEO" esta petición me devuelve un objeto json muy grande anidado con muchos objetos dentro (cada objeto es una orden de compra realizada por un cliente), una orden tiene alrededor de 100 campos o tributos, o sea es bastante complejo pero bien estructurado (incluye por ejemplo nombre del comprador, el producto, la dirección de envío en fin muchos atributos), la petición POST me devuelve las ordenes de los últimos 5 meses, si tenemos en cuenta que un día cualquiera hay una aproximado de 6mil ordenes, y un día como navidad pueden haber por alta demanda 20mil ordenes, notarás que ya se trata de un ambiente de Bigdata. Cabe recalcar aquí que esta petición post no me devuelve las ordenes de los últimos 5 meses en una sola petición, en realidad la api del "CEO" me devuelve solo 100 órdenes por un tema de rendimiento, por lo que hay que hacer una especie de paginación de 100 en 100 hasta que me de todas las ordenes de 5 meses atrás. La api tiene un campo "total de orden" y se sabe hasta cuando habría que iterar.

Sigamos con el escenario general. Se hace la petición POST y cada una de las ordenes quiero insertarla en Bigquery. O sea, hay que diseñar una tabla de Bigquery que tenga como columnas los atributos internos de una orden. Una orden tiene un campo que es un identificador, como este proceso se va a hacer todos los días la idea es que las ordenes que sean nuevas o sea que aún no existan en bigquery hay que insertarlas, pero si ya existen solo hay que actualizarlas.

Cuando dicha orden ya está insertada o actualizada (ya sea una u otra, siempre debe ser una u otra) entonces tengo que notificarla al "CEO" a través de otra petición POST "diciéndole" que ya tengo esa orden en bigquery. La idea de hacer esta notificación exitosa es asegurarme de que no hubo ningún problema durante un proceso intermedio de transformación y que efectivamente me llegó la orden a bigquery y la tengo lista para hacer análisis y machine learning en el futuro.

Cual podría ser una solución costo efectiva en GCP

Análisis de la Propuesta de Solución:

1. Extracción de Datos y Almacenamiento Intermedio

Cloud Scheduler & Cloud Functions: Utiliza Cloud Scheduler para activar una Cloud Function diariamente a las 2 am. Esta función ejecutará el script de extracción de datos.

Paginación y Extracción de Datos: La Cloud Function realizará las peticiones POST paginadas al sistema "CEO". Para manejar el volumen de datos, puedes considerar almacenar los resultados intermedios en un formato eficiente como JSON o Avro.

Google Cloud Storage (GCS): Almacena los datos extraídos en GCS. Elige un formato de archivo adecuado (CSV, JSON, Avro, o Parquet) según tus necesidades de procesamiento y análisis posterior.

2. Procesamiento y Carga en BigQuery

Apache Beam con Google Cloud Dataflow: Utiliza Apache Beam para crear un pipeline de procesamiento de datos que se ejecutará en Dataflow. Este pipeline realizará las siguientes operaciones:

Leer los datos desde GCS.

Transformar los datos si es necesario (por ejemplo, normalizar campos, calcular agregaciones, etc.).

Verificar si cada orden ya existe en BigQuery y decidir si insertar o actualizar.

Cargar los datos en la tabla original de BigQuery.

Realizar las transformaciones necesarias y cargar los datos en una segunda tabla transformada en BigQuery.

3. Notificación de Éxito

Cloud Functions para Notificación: Una vez que los datos estén cargados en BigQuery, usa otra Cloud Function para notificar al sistema "CEO" a través de una petición POST por cada orden procesada.

4. Orquestación de Procesos

Cloud Composer (Apache Airflow): Utiliza Cloud Composer para orquestar todo el flujo de trabajo. Esto incluye:

La activación de la Cloud Function de extracción de datos.

La ejecución del pipeline de Dataflow.

La activación de la Cloud Function de notificación.

Razones para Esta Arquitectura

Escalabilidad y Flexibilidad: Esta arquitectura es escalable para manejar el alto volumen de datos y las variaciones en el número de órdenes (como en temporadas altas).

Costo-Eficiencia: El uso de servicios gestionados como Cloud Functions, Dataflow y BigQuery optimiza los costos, ya que pagas solo por los recursos que utilizas.

Fiabilidad y Mantenibilidad: Al utilizar servicios de GCP, te beneficias de la alta disponibilidad y la facilidad de mantenimiento.

Automatización y Orquestación: Cloud Composer facilita la gestión de dependencias y la secuencia de tareas, asegurando que el flujo de trabajo se ejecute sin problemas.

Cloud Run VS Cloud Functions o App Engine

Para entender por qué Cloud Run puede ser una mejor opción que Cloud Functions o App Engine en tu caso, es importante considerar las características específicas de tu tarea y las capacidades de cada plataforma. Vamos a compararlas en términos de flexibilidad, tiempo de ejecución, escalabilidad y costos.

Google Cloud Run

**Ventajas:**

Flexibilidad en Tiempos de Ejecución: Cloud Run permite tiempos de ejecución más prolongados en comparación con Cloud Functions. Puede manejar tareas que duren hasta 15 minutos o más, dependiendo de la configuración, lo cual es ideal para tu caso de uso.

Contenerización: Cloud Run utiliza contenedores, lo que te ofrece flexibilidad para incluir cualquier biblioteca o dependencia necesaria sin preocuparte por el entorno de ejecución.

Escalabilidad Automática: Escala automáticamente y puede escalar a cero cuando no hay solicitudes, lo que significa que no pagas por el tiempo de inactividad.

Costos: Solo pagas por el tiempo de ejecución de las solicitudes, lo que puede ser más económico para tareas que se ejecutan una vez al día.

**Desventajas:**

Requiere un conocimiento básico de contenedores y Docker.

Cloud Functions

**Ventajas:**

Facilidad de Uso: Es muy sencilla de configurar y usar, ideal para tareas pequeñas y ligeras.

Costos: Solo pagas por el tiempo de ejecución, lo que es ideal para funciones que se ejecutan en respuesta a eventos.

**Desventajas:**

Límite de Tiempo: Tiene un límite de tiempo de ejecución máximo (9 minutos), lo que no es adecuado para tu tarea que puede durar 15 minutos.

Menos Flexibilidad: Menos flexibilidad en términos de entorno de ejecución en comparación con los contenedores.

Google App Engine

**Ventajas:**

Fácil Despliegue de Aplicaciones Web: Excelente para aplicaciones web que requieren escalabilidad automática.

Gestión de Infraestructura: Gestiona la infraestructura por ti, lo que reduce la sobrecarga de mantenimiento.

**Desventajas:**

Costo y Recursos: Aunque escala automáticamente, puede ser más costoso mantener una instancia en ejecución constantemente, especialmente si la tarea solo necesita ejecutarse una vez al día.

Tiempo de Ejecución: Los tiempos de ejecución prolongados podrían no ser ideales para App Engine, que está diseñado más para responder a solicitudes web rápidas.

**En términos de costos:**

Cloud Run: Solo pagas por el tiempo de ejecución real. Si tu servicio está inactivo, no incurre en costos. Esto lo hace ideal para tareas que se ejecutan en horarios específicos y no constantemente.

Cloud Functions: Similar a Cloud Run en términos de costos, pero con las limitaciones de tiempo ya mencionadas.

App Engine: Puede incurrir en costos más altos, especialmente si necesita estar ejecutándose continuamente o manejar cargas de trabajo impredecibles.

**Conclusión Parcial**

Para una tarea que se ejecuta diariamente y puede durar hasta 15 minutos, Cloud Run es probablemente tu mejor opción debido a su flexibilidad, capacidad para manejar tiempos de ejecución más largos y un modelo de precios que solo te cobra por el tiempo de ejecución real. Además, te brinda la flexibilidad de trabajar con contenedores, lo que significa que puedes configurar tu entorno exactamente como lo necesitas.

**Propuesta de solución**

1. Una Cloud Run contiene una app Python que se encarga de hacer las peticiones al CEO de forma paginada y colocar cada objeto json extraído en GCS, (en un archivo. json correspondiente al día solicitado) que tiene todos los objetos json extraídos de las peticiones. (Se genera un archivo por día).
   1. Se utilizan técnicas de CI/CD para este despliegue. En Editor de Código de la Cloud Shell. Una vez probado el código localmente usando un entorno virtual de Python para aislar las dependencias. Se hace un Push a una rama por ejemplo Staging (repositorio creado previamente en Cloud Sources Repositories ver [extract-ceo-app-repository](https://source.cloud.google.com/warehousev2-ce0-alas/extract-ceo-app-repository?authuser=1) .
   2. Al hacer este Push se dispara un trigger de Cloud Build (herramienta CI/CD de GCP) que lee el archivo Dockerfile y procede a construir una imagen. Dicha imagen es almacenada en Container Registry automáticamente.
   3. Desde Cloud Runner se importa esta imagen y se despliega. (Esta última parte puede ser manual o bien automáticamente leyendo alguna actualización de una rama del repositorio). Cabe destacar que esta url proporcionada puede ser pública o protegida (requerir autenticación, variante escogida en este caso).
2. Se realizan 5 Dataflows. El objetivo de cada uno es desagregar campos json anidados que están dentro de cada objeto extraído y convertirlo en una tabla distinta de bigquery (se usan transformaciones FlatMap en la que se permite a partir de un solo elemento, que pueda ser transformado en varios elementos de salida, esto es así porque un subcampo json es un array de varios objetos json a su vez).

Primero se desarrolla el pipeline de Beam en Jupyter Notebook (Workbench de Dataflow), esto es así para aprovechar las capacidades de exploración de datos, visualización y pruebas intermedias que son útiles durante del desarrollo del pipeline, además se usa **DirectRunner** para ejecutar este pipeline y ver un resultado progresivo.

Una vez que se tiene una línea base del pipeline entonces se procede a traspasarlo (exportando a .py) a un ambiente de desarrollo mas adecuado para continuar con las respectivas pruebas, desarrollo y despliegue.

Una vez en el Code Editor de la Shell de GCP se procede a probar el desarrollo en un entorno virtual de Python para asegurar el funcionamiento con las dependencia adecuadas (Ojo: las dependencias y versiones usadas por defecto en el Jupyter Notebook serían las mismas que son importadas en el entorno virtual siempre y cuando no sean parte de la propia librería de python).

Luego se procede al diseño de las Pruebas Unitarias usando el Framework **UnitTest** de Python. Consiste en el desarrollo de múltiples pruebas unitarias que posteriormente se deben ejecutar cuando se despliega el proyecto (pipeline).

Entonces se procede a diseñar un flujo de **CI/CD** a través del archivo **cloudbuild.yaml** y la herramienta **Cloud Build**. El objetivo es que cuando se haga un push a una rama por ejemplo Staging en Cloud Source Repositories, sea ejecutado el archivo cloudbuild.yaml (se hace configurando un trigger de Cloud Build que lea el .yaml). Este .yaml define una serie de pasos en orden durante el despliegue al repositorio priorizando la ejecución automática de pruebas unitarias y si pasan correctamente continua con un segundo paso que sería copiar este pipeline de beam (fichero .py) al bucket GCS correspondiente al DAG de Airflow (Composer) ya que la tarea programada de Composer siempre estará leyendo de este archivo en esta ubicación para ejecutar la tarea diaria. Ver repositorio [alas\_dataflow\_1\_repository](https://source.cloud.google.com/warehousev2-ce0-alas/alas_dataflow_1_repository?authuser=1) .

1. En este punto queda desarrollar todo el esquema de las tablas y procedimientos almacenados dentro de Bigquery, (los esquemas de las 5 tablas para probar los 5 dataflows deberían existir). Se procede a hacer una serie de procedimientos almacenados que son transformaciones a partir de las 5 tablas iniciales provenientes de los 5 dataflows. Transformaciones que requieren el uso de **Update y Delete** continuadamente hasta llegar a las tablas delivery\_order\_master (histórica) y delivery\_order\_work.
2. Orquestación a través de Composer, se crea una DAG (**DAG\_cloudrun\_dataflow\_sp** subido al bucket del DAG correspondiente) que básicamente inicia una serie de tareas a las 2am comenzando por la llamada (autenticada) a la Cloud Run (usando PythonOperator). La misma desencadena la lectura de ordenes y guarda en GCS los **archivos json** de cada día de los últimos 5 meses. Luego desencadena los 5 dataflows (usando BeamRunPythonPipelineOperator) en orden y por último todos los procedimientos almacenados (BigQueryExecuteQueryOperator) en el orden correcto.

**Conclusión:**

Debajo de Google Cloud Composer hay un clúster de Kubernetes. Google Cloud Composer utiliza Google Kubernetes Engine (GKE) para orquestar y gestionar los contenedores que ejecutan Apache Airflow y sus componentes relacionados. Este enfoque aprovecha la escalabilidad, la gestión de la infraestructura y las capacidades de auto-curación de Kubernetes, proporcionando una plataforma robusta y escalable para la orquestación de flujos de trabajo.

Por ello y sin embargo el uso de Cloud Composer termina siendo **costoso** y el paradigma de creación de 5 dataflows seguidos provoca lentitud en el flujo de trabajo ya que Composer tiene que iniciar primeramente la infraestructura de cada Dataflow antes de lanzar un job sobre él. El proceso demora alrededor de 2 horas lo que no cumple con el criterio de **eficiencia/rendimiento**.