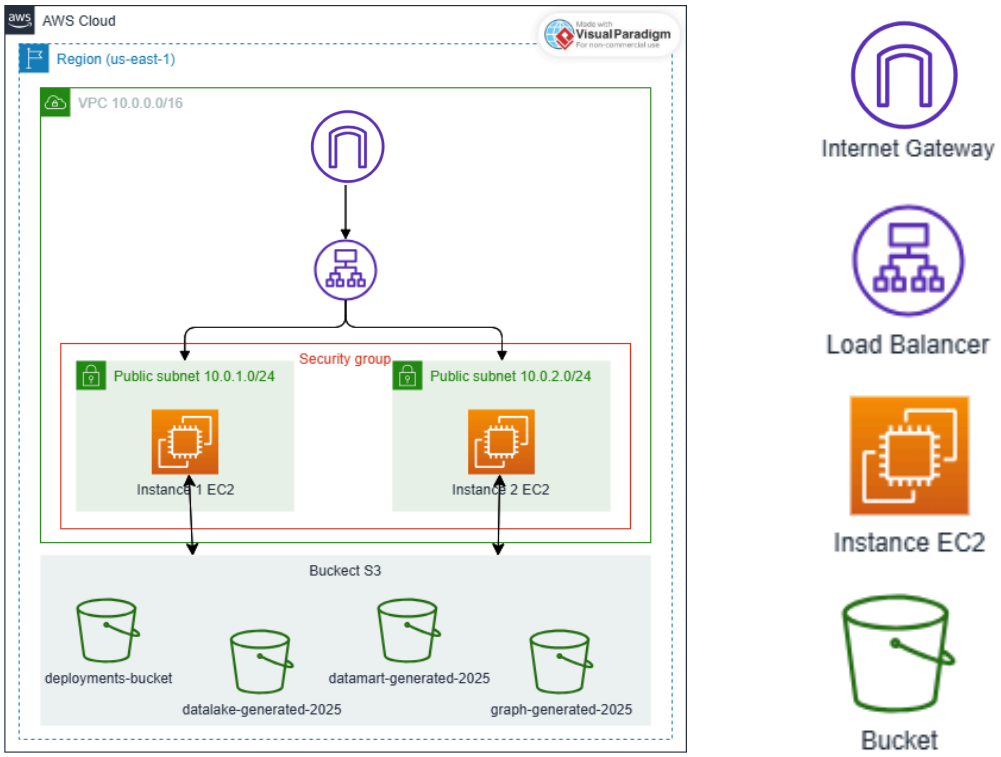
Explicación arquitecturas

Diagrama AWS EC2



Esta arquitectura en AWS está diseñada para desplegar una aplicación distribuida utilizando instancias EC2, un balanceador de carga (ALB) y buckets S3 para almacenar los scripts y los datos generados. Todo se implementa en la región us-east-1 dentro de una VPC que incluye dos subredes públicas, cada una ubicada en diferentes zonas de disponibilidad para mejorar la tolerancia a fallos y la disponibilidad del sistema.

La **red privada virtual (VPC)** abarca un rango de direcciones IP 10.0.0.0/16 y cuenta con un Internet Gateway que permite la conexión a internet. En el interior hay dos subredes públicas: una con el bloque 10.0.1.0/24 y la otra 10.0.2.0/24. Cada una está configurada para hospedar una instancia EC2 y está asociada a una tabla de rutas que direcciona el tráfico saliente al internet Gateway.

El **grupo de seguridad** asociado a las instancias y al balanceador de carga permite tres tipos de conexiones entrantes:

- **Puerto 22 (SSH):** Permite acceso remoto mediante SSH, lo que permite realizar tareas de administración y mantenimiento en las instancias
- **Puerto 80 (HTTP):** Corresponde al tráfico HTTP estándar, permite que los usuarios puedan acceder a la API a través de una URL pública generada por el balanceador de carga.
- **Puerto 5000:** Es utilizado para la comunicación interna entre el balanceador de carga y la aplicación Flask que corre en las instancias EC2.

El **balanceador de carga (ALB)**, configurado para recibir solicitudes desde el exterior y distribuirlas a las instancias en ambas subredes. Este ALB utiliza reglas definidas en un listener para enrutar el tráfico recibido en el puerto 80 hacia el puerto 5000 de las instancias EC2. Además, monitorea el estado de las instancias mediante peticiones al puerto 5000, comprobando la ruta raíz (/) cada 30 segundos para verificar que la aplicación responde correctamente. Si una instancia no pasa la revisión de estado, el balanceador deja de enviarle tráfico y evita interrupciones en el servicio.

Las instancias EC2 ejecutan un script de inicio que instala dependencias (boto3 y Flask), descarga scripts desde el bucket S3 y ejecuta las siguientes tareas:

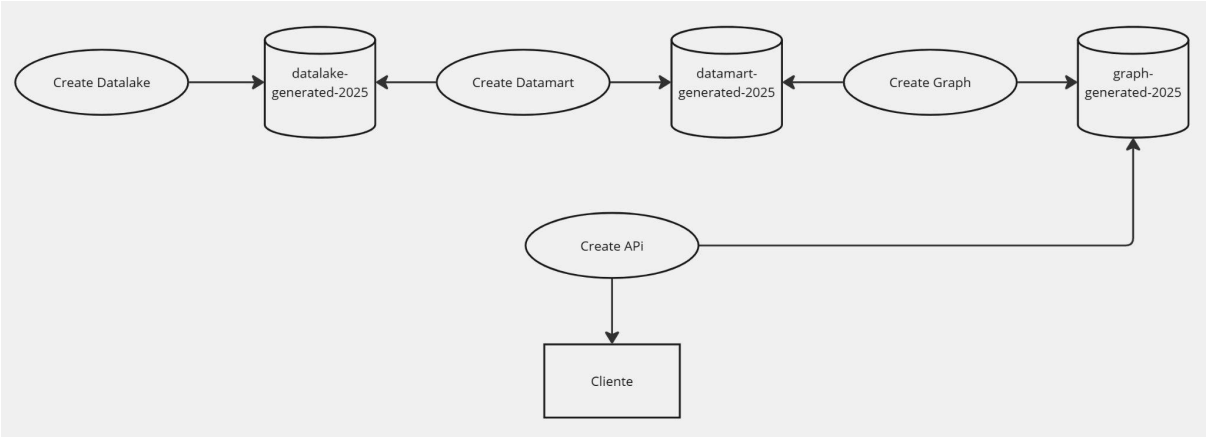
1. **Crear un datalake** con documentos descargados desde Project Gutenberg.
2. **Generar un datamart** con diccionarios de palabras filtradas de 3, 4 y 5 letras.
3. **Construir un grafo** conectando palabras que difieren en una letra.
4. **Lanzar la API Flask** que expone los datos procesados a través de distintos endpoints.

Por otro lado, el bucket S3 deployments-bucket456 contiene todos los scripts necesarios para las tareas de despliegue y procesamiento de datos. Esto permite un despliegue dinámico, ya que las instancias EC2 descargan automáticamente los archivos al iniciarse.

De esta manera el flujo de trabajo es el siguiente:

1. **Carga de scripts en S3:** Los archivos de Python son subidos al bucket deployments-bucket456.
2. **Creación de infraestructura:** Se crea la VPC, subredes, tabla de rutas y grupo de seguridad.
3. **Despliegue del balanceador de carga:** Se configura el ALB y el target group.
4. **Lanzamiento de instancias EC2:** Se inician instancias en cada subred, asociadas al balanceador.
5. **Ejecución de scripts:** Cada instancia descarga y ejecuta los scripts desde el bucket S3, creando nuevos bucket con información.
6. **Consulta:** El balanceador de carga expone un DNS público para acceder a la API.

Diagrama de negocio



El diagrama presentado refleja la estructura de negocio planteada en el proyecto el cual sigue el siguiente flujo:

1. **Módulo Create Datalake**
   El proceso comienza con este módulo, el cual va a crear un bucket llamado *datalake-generated-2025*, para luego descargar documentos desde Project Gutenberg y guardarlos en el bucket.

2. **Módulo Create Datamart**
   Este módulo crea un bucket llamado *datamart-generated-2025*. Acto seguido va a procesar los archivos que se encuentran dentro del bucket *datalake-generated-2025*. Durante el preprocesamiento se van a eliminar las secuencias de letras poco comunes y números. A continuación, las palabras se organizan según su longitud (tres, cuatro o cinco letras) y se almacenan en diferentes archivos dentro del Datamart. Estos archivos siguen un formato que incluye la palabra, seguida de dos puntos y la cantidad de veces que aparece en el texto, por ejemplo: ***the: 400***.

3. **Módulo Create Graph**
   Este módulo crea un bucket llamado *graph-generated-2025*. Para luego, tomar los datos de *datamart-generated-2025* y se construir un grafo. De esta manera cada nodo representa una palabra, y se crean conexiones (aristas) entre aquellas palabras que difieren en una sola letra. El peso de cada conexión se calcula restando la frecuencia de aparición de una palabra con la de la otra palabra conectada. Este valor refleja la diferencia en la cantidad de veces que cada palabra aparece en el texto procesado, y se utiliza como un indicador del "peso" de la conexión entre ambas palabras. Este archivo sigue un formato que incluye la primera palabra, la segunda palabra y el peso, por ejemplo: ***the she 1716***

4. **Módulo Create API**
   Finalmente, se implementa una API para interactuar con el grafo creado. Entre las consultas que pueden realizarse destacan: la búsqueda del camino más corto mediante el algoritmo de Dijkstra, el cálculo del camino más largo, la identificación de nodos aislados, la detección de nodos con un alto grado de conectividad, el cálculo de todos los caminos y la detección de cluster. Esta API se despliega en AWS mediante un API Gateway, lo que permite su acceso desde internet de forma segura.

Decisiones de diseño

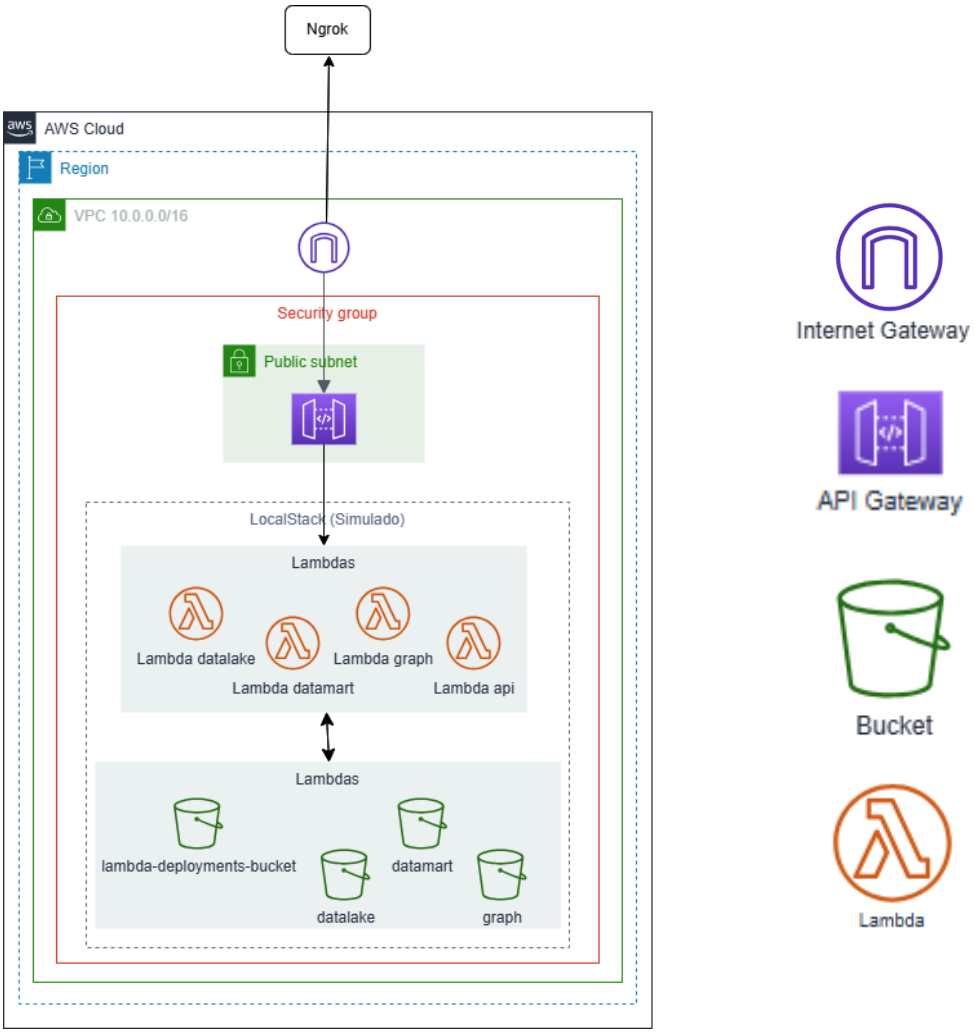
En la versión 1.0 de nuestro proyecto **GraphWords**, la arquitectura final implementada en AWS se basó en instancias EC2. Sin embargo, el desarrollo inicial comenzó con una estructura basada en funciones Lambda. Durante las primeras etapas, enfrentamos problemas relacionados con permisos en AWS que impedían la creación y ejecución de las funciones Lambda directamente en la nube. Esto nos obligó a simularlas utilizando herramientas como LocalStack. Al hacerlo, observamos que gran parte del proyecto quedaba restringida a un entorno local, alejándonos del objetivo principal de implementar completamente en la nube de AWS.

Por esta razón decidimos utilizar instancias EC2 no solo para mantener el proyecto completamente en AWS, sino también porque nos ofrecían mayor flexibilidad y control en la configuración del entorno. A diferencia de las funciones Lambda, las instancias EC2 permiten personalizar los recursos según las necesidades específicas del proyecto, como la instalación de dependencias adicionales, ajustes de hardware y la capacidad de manejar procesos más complejos sin las restricciones que suelen tener las funciones serverless. Este enfoque también facilita la supervisión y mantenimiento del sistema, asegurando una mayor estabilidad y adaptabilidad para futuras actualizaciones o incrementos en la carga de trabajo.

Para manejar las consultas de manera eficiente, optamos por implementar un **Load Balancer** que distribuye las solicitudes entrantes entre dos instancias EC2. Esta configuración fue suficiente para el volumen y tamaño del proyecto, ya que no era necesario añadir más instancias dada la carga estimada de trabajo. El balanceador de carga asegura que las consultas se gestionen de manera simultánea, optimizando el rendimiento general del sistema.

Inicialmente, consideramos la implementación de un sistema de autoscaling que ajustara automáticamente el número de instancias EC2 en función de la demanda. Sin embargo, debido a limitaciones en los permisos de AWS, no fue posible llevarlo a cabo en esta versión del proyecto. La inclusión de autoscaling hubiera permitido una mayor flexibilidad y robustez del sistema al adaptarse dinámicamente a las fluctuaciones de la carga de trabajo. En futuras versiones, planeamos habilitar esta funcionalidad para mejorar aún más la escalabilidad y garantizar un rendimiento óptimo en cualquier escenario.

Diagrama AWS inicial con Lambda



Esta arquitectura está diseñada para desplegar una aplicación distribuida simulada localmente utilizando funciones Lambda, LocalStack, y un túnel seguro proporcionado por Ngrok. La infraestructura sigue el modelo de AWS, simulando componentes dentro de una VPC con acceso externo.

La **red privada virtual (VPC)** abarca un rango de direcciones IP 10.0.0.0/16 y cuenta con un Internet Gateway que permite la conexión a internet. En el interior hay una subred pública: una con el bloque 10.0.1.0/24. Esta subred contiene un API Gateway simulado mediante Ngrok que crea un túnel que expone un endpoint público y redirige el tráfico HTTP hacia LocalStack.

El **grupo de seguridad** permite tráfico en el puerto 80 (HTTP) para las solicitudes desde internet. Asegurando que solo las funciones Lambda tengan acceso al sistema de almacenamiento.

Las lambdas ejecutan un script de inicio que instala dependencias y descargan scripts desde el bucket S3 y ejecuta las siguientes tareas:

5. **Crear un datalake** con documentos descargados desde Project Gutenberg.
6. **Generar un datamart** con diccionarios de palabras filtradas de 3, 4 y 5 letras.
7. **Construir un grafo** conectando palabras que difieren en una letra.
8. **Lanzar la API Flask** que expone los datos procesados a través de distintos endpoints.

Por otro lado, el bucket S3 lambda-deployments-bucket contiene todos los scripts necesarios para las tareas de despliegue y procesamiento de datos.

Para todo este proceso se utiliza **Ngrok** crea un puente seguro entre internet y tu aplicación local, generando una URL pública que permite acceder a la API Flask sin necesidad de abrir puertos ni exponer tu red. Recibe las solicitudes en la URL pública, las redirige automáticamente al puerto 5000 de tu máquina local donde corre la API, y luego devuelve la respuesta al cliente de manera segura utilizando un túnel HTTPS que protege la transmisión de datos. En este trabajo, Ngrok se utiliza para simular el comportamiento de un API Gateway en AWS, permitiendo que la API local expuesta por LocalStack sea accesible desde internet para pruebas externas sin necesidad de desplegar la infraestructura en la nube, facilitando así la validación de las funciones Lambda y la interacción segura con los datos procesados

De esta manera el flujo de trabajo es el siguiente:

7. **Carga de scripts en S3:** Los archivos de Python son subidos al bucket lambda-deployments-bucket.
8. **Creación de infraestructura:** Se crea la VPC, subred, tabla de rutas y grupo de seguridad.
9. **Lanzamiento de instancias Lambda:** Se inician las lambdas.
10. **Ejecución de scripts:** Cada instancia descarga y ejecuta los scripts desde el bucket S3, creando nuevos bucket con información.

11. **Consulta:** La API devuelve los resultados (por ejemplo, el camino más corto entre palabras) al cliente.