**Virtualización en la Nube**
**Universidad Politécnica Salesiana**
**Computación - Periodo 67**

**Integrantes:** Karen Ortiz - Andrés Encalada

**Objetivo de la práctica:**

Se debe diseñar, desplegar y documentar una arquitectura distribuida de tres capas en una plataforma en la nube (AWS, Google Cloud o Azure), incorporando servicios de almacenamiento compartido y balanceo de carga.
La solución simulará el backend de un portal académico donde se gestionan recursos multimedia, generados o cargados por los usuarios del sistema.

**Plataforma: Google Cloud**

Esta práctica fue realizada haciendo uso de la plataforma Google Cloud, la cual proporciona las herramientas necesarias para cumplir con los objetivos propuestos en el enunciado. Entre los servicios destacados que fueron de utilidad para la práctica tenemos creación de máquinas virtuales, balanceadores de carga y autoscaling, creación de redes y reglas de Firewall, almacenamiento compartido y bases de datos.

*Desarrollo de la actividad*

**Parte 1: Preparación y aprovisionamiento del entorno**

1. **Infraestructura base en la nube**

- **Instancias Virtuales**

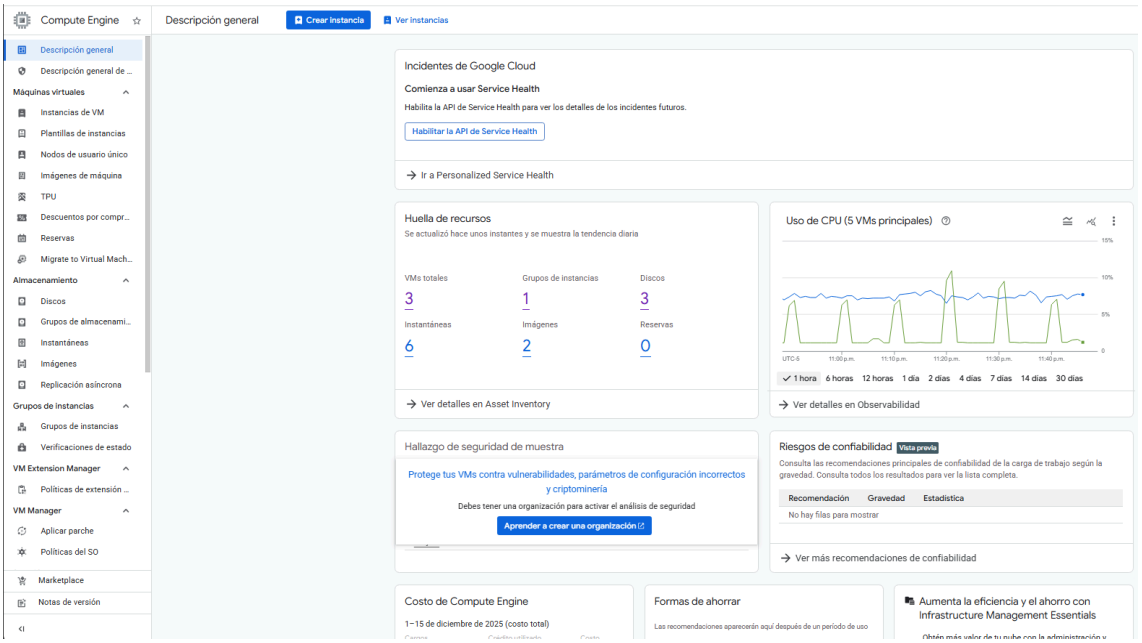
Todas las instancias fueron creadas con la misma configuración, incluida las máquinas generadas por el balanceador de carga. Por lo que todas usan un sistema operativo Ubuntu 22.04 LTS x86-64 y están alojadas en el servidor ‘us-central1-c’. A continuación se presentan las instancias creadas inicialmente.

- **ui-server:** Instancia que posee el servidor de la interfaz gráfica, pertenece a la red ‘subred-publica’.
- **api-server:** Instancia que posee la API, la cual conecta al frontend con la base de datos y el almacenamiento compartido, pertenece a la red ‘subred-´privada’. Después de generar una plantilla está instancia es detenida y apagada.

- **Creación de componentes**

Para crear una instancia debemos dirigirnos a la sección Compute Engine, Instancias de VM y posteriormente, Crear una instancia. La configuración de cada instancia dependerá directamente de las necesidades del proyecto. Además, en la sección Compute Engine, encontramos otras funciones, como grupos de instancias, plantillas de instancias e imágenes de instancias, funciones esenciales para el funcionamiento del autoescalamiento y balanceo de carga.

*Fig. 1.* **Panel principal de la sección Compute Engine**




## 2. Configuración de seguridad y redes

Se creó una red llamada ‘red-práctica’, la cual contiene 2 subredes, ‘subred-publica’ y ‘subred-privada’

- Acceso HTTP público al servidor UI.

La subred pública permite el acceso HTTP a los usuarios desde cualquier dirección IP, garantizando un ingreso público al servidor U. Para esto se habilitó el puerto 80 con el protocolo TCP.

- Acceso restringido API → DB mediante puertos específicos.

Para la comunicación interna se definió una regla de firewall que permite la comunicación desde los destinos 10.0.0.0/8 (Máquinas virtuales) y 10.129.0.0/23 (Proxy), además, se definieron puertos específicos para lograr una comunicación entre los servicios. TCP: 22,3000,5432,111,2049,80, UDP: 2049,111 y el protocolo ICMP para habilitar el servicio ping. A continuación, se presenta un desglose de los puertos y su función.

*Tabla 1.* **Explicación de puertos habilitados**

| Protocolo | Puerto | Función |
|---|---|---|
| TCP | 22 | Usado para ingresar mediante SSH |
| | 3000 | Es usado por el servidor de backend para escuchar las peticiones. |
| | 5432 | Se usa para que la API pueda comunicarse con la base de datos (PostgreSQL) |
| | 80 | Usado para la comunicación con el servidor web. |
| TCP/UDP | 2049 | Usado para almacenar los archivos en Filestore |
| | 111 | Utilizado por los servicios para poder obtener la ruta del almacenamiento compartido |
| ICMP | - | Protocolo utilizado para habilitar el comando ping. |

- Acceso del API Instance al servicio de almacenamiento compartido.
  Este acceso se da gracias al uso del puerto 111 y 2049, explicados en la Tabla 1.
- SSH solo para direcciones IP autorizadas.

Para esto se estableció un rango de direcciones 35.235.240.0/20, usando el protocolo TCP en el puerto 22, con esto logramos que solo los administradores autenticados en la consola de Google puedan establecer una conexión.

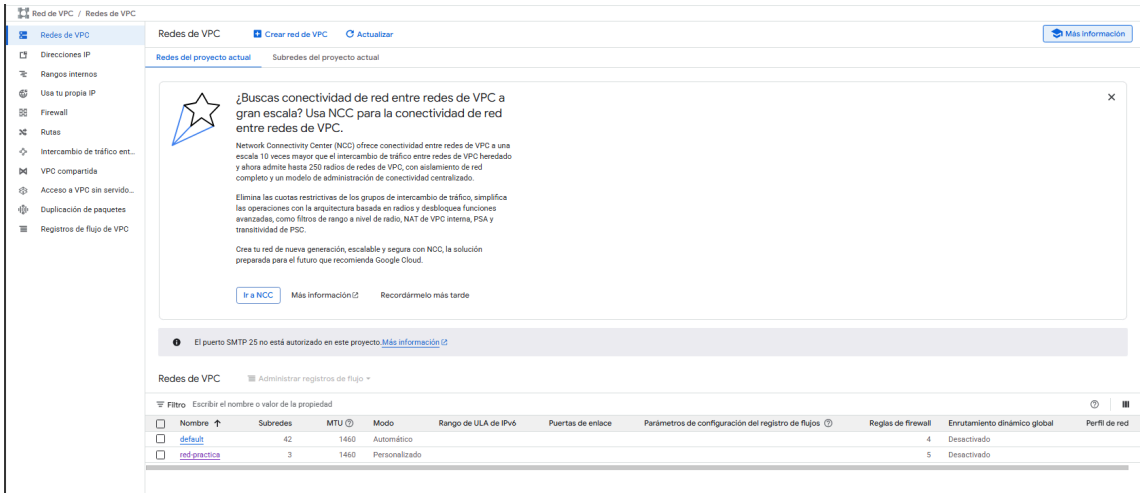
- Etiquetado, agrupamiento y subredes según la plataforma escogida.

Se utilizaron las etiquetas ‘allow-health-check’ y ‘allow-ssh’ para autorizar el tráfico entre las instancias. De esta forma se logra un agrupamiento en las instancias, a las que se les aplicaron leyes de firewall donde se establecen las subredes, direcciones IP y puertos con los que se pueden comunicar.

- **Creación de componentes**

Para crear las redes y subredes, debemos dirigirnos a la sección Redes de VPC, donde podremos crear nuestra red principal, sus subredes, y las políticas y reglas de Firewall.

Fig. 2. **Panel principal de la sección Redes de VPC**



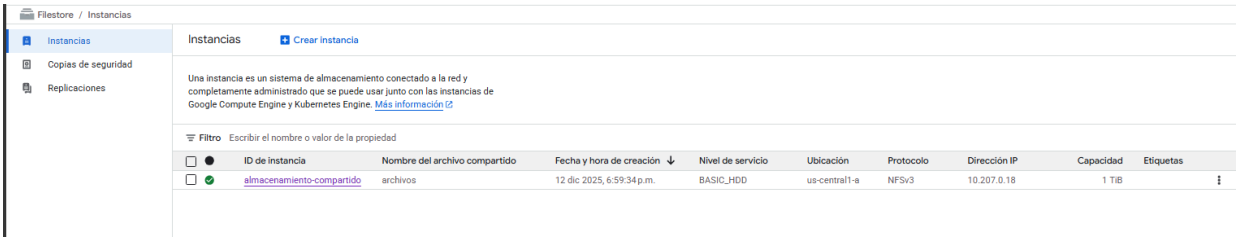

### 3. Almacenamiento compartido

Se configuró el servicio Filestore integrado en la plataforma Google Cloud, posteriormente se creó un directorio de montaje local en '/mnt/storage', el cual sirve como interfaz de acceso al disco remoto. Finalmente, se editó el archivo '/etc/fstab' para garantizar que cualquier nueva instancia creada por el Autoescalador monte el disco automáticamente al arrancar.

- **Creación de componentes**

Para crear una instancia de FileStore, debemos buscar la sección del mismo nombre. Al crear una instancia podremos configurar diferentes parámetros, como la capacidad, la región y la red a la que pertenece, entre otros.

*Fig 3.* **Panel principal de Filestore**




## Parte 2: Implementación de la plataforma

### 1. Capa de Interfaz

Esta capa actúa como el punto de entrada para los usuarios finales, proporciona una interfaz gráfica para interactuar con los servicios del sistema. Se implementó siguiendo el modelo SPA (Single Page Application) con React.js.

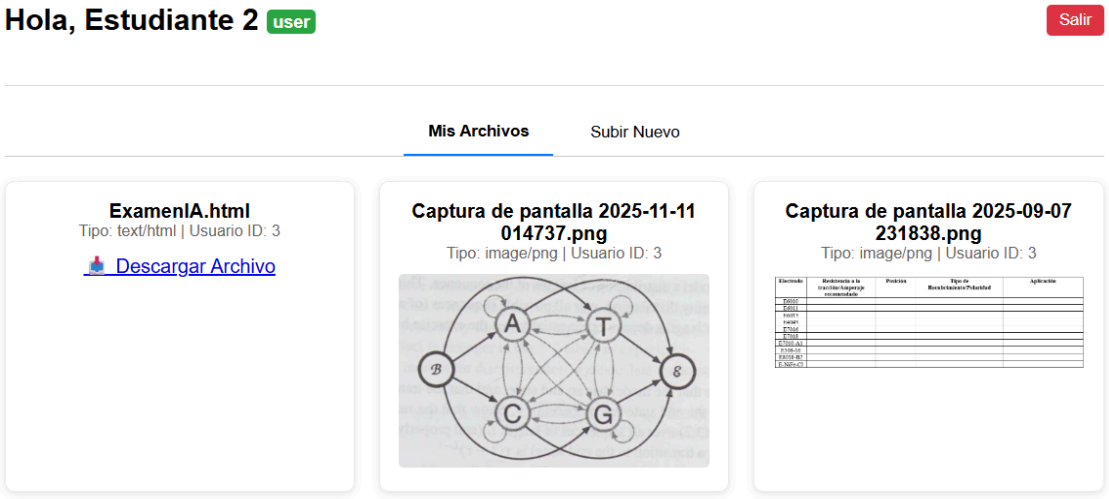
La interfaz actúa como como un consumidor de la API RESTful, por medio del protocolo HTTP asíncrono y se realiza el intercambio de información en formato JSON. Se divide en dos vistas lógicas:

- **Módulo de autenticación:** permite al usuario iniciar sesión y registrar usuarios (el rol de admin tiene visualización completa a todos los archivos subidos por los usuarios). Además de poder visualizar errores (credenciales inválidas y usuario creado).

*Fig 4.* **Módulo de autenticación**


- **Panel Principal:** Visualización en cuadrícula de los archivos subidos con previsualización y opción de descarga.

*Fig 5.* **Dashboard con los archivos subidos por usuarios**



- Subida de archivos: Formulario para seleccionar la categoría y el archivo a subir.

*Fig 6.* **Sección subir archivos**

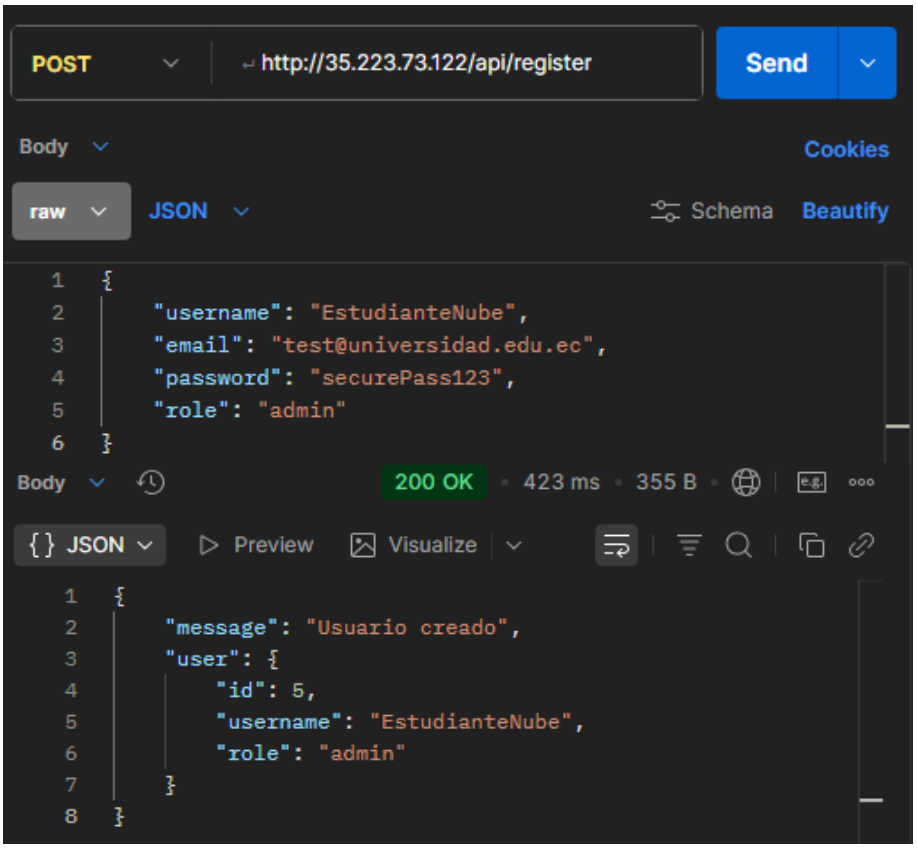

## 2. Capa de Servicios

Se desarrolló un conjunto de endpoints REST, los cuales sirven para registrar usuarios, subir, listar y obtener metadatos de los archivos.

Para comprobar el funcionamiento de los endpoints se realizaron pruebas en Postman. A continuación se presentan los resultados de cada endpoint.

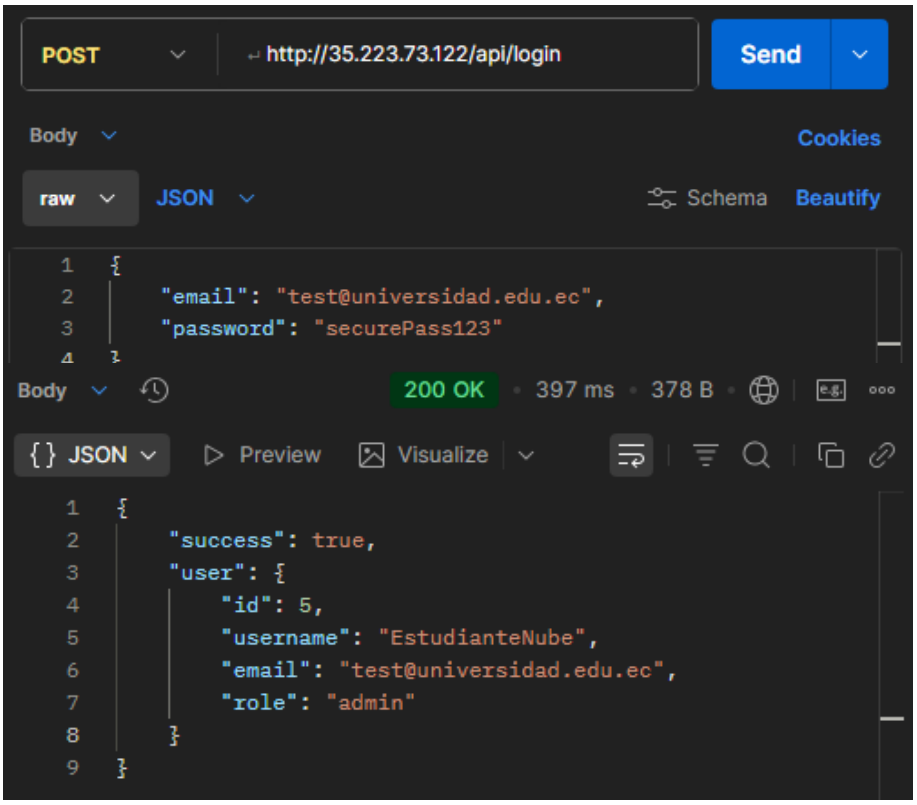
- Conexión API a base de datos

*Fig 7.* **Operación Post registrando un usuario**



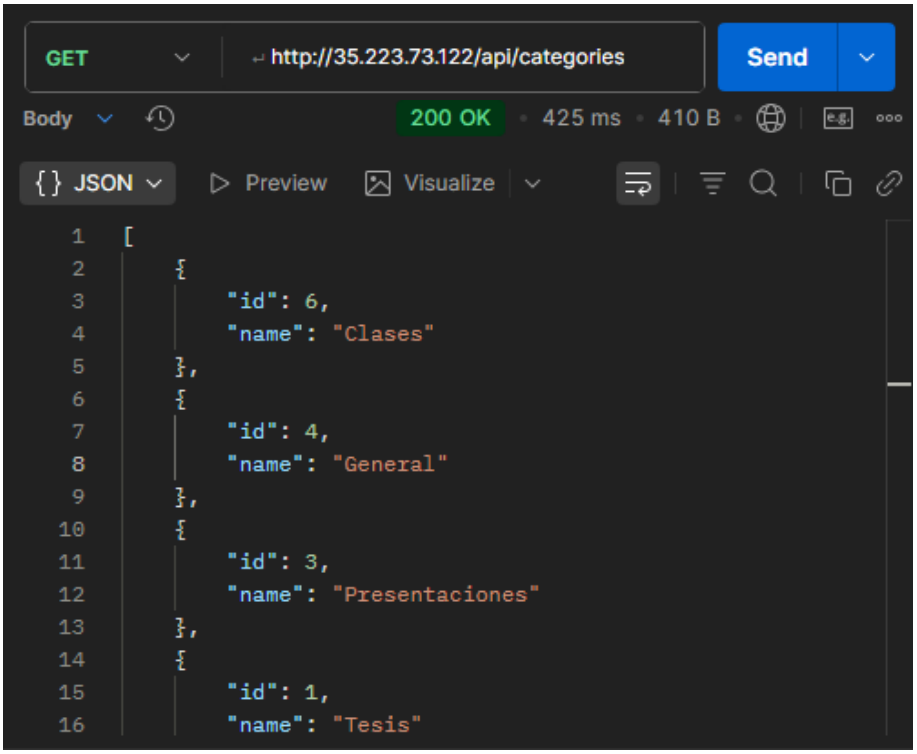
- Prueba de Login

*Fig 8.* **Operación Post para iniciar sesión**



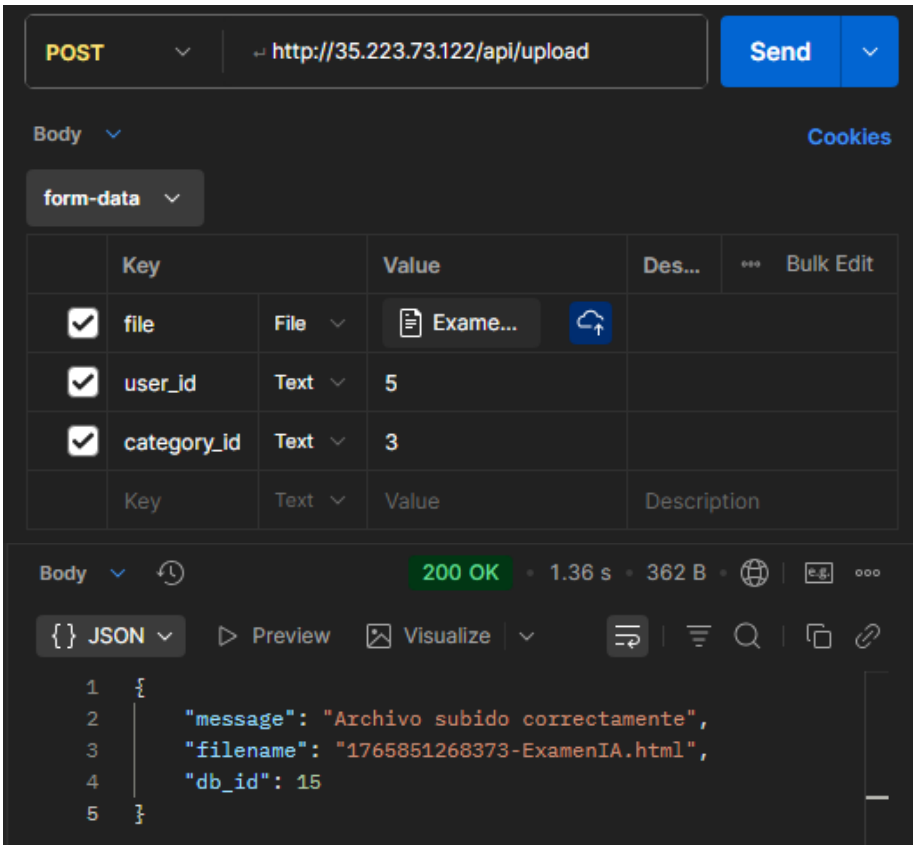

- Obtención de datos

*Fig 9.* **Operación Get para obtener categorías**



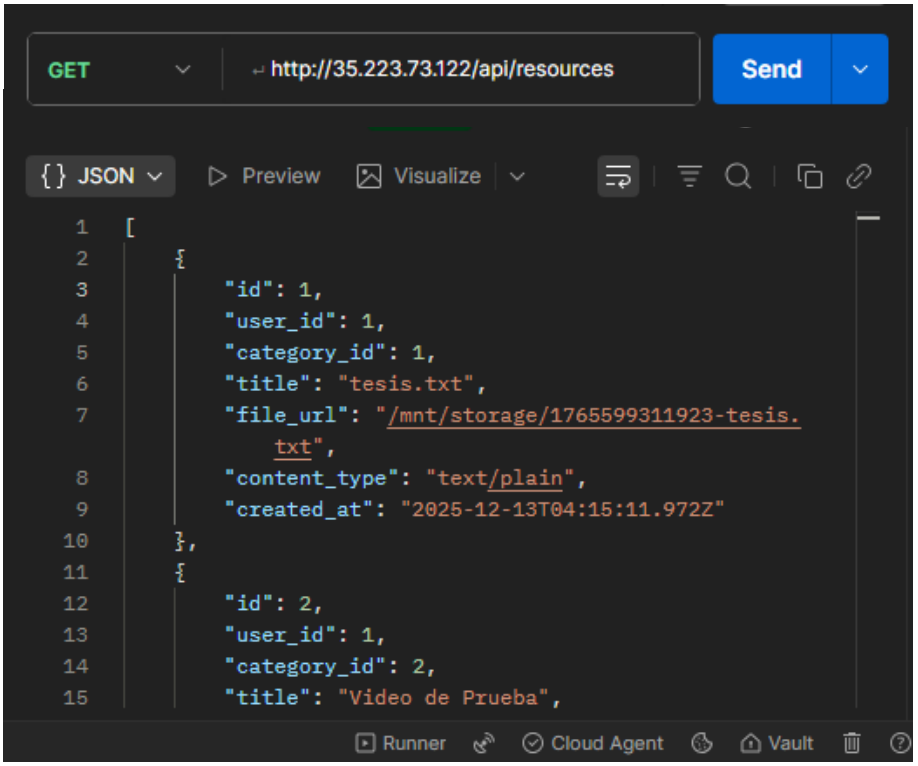
- Subida de archivos

*Fig 10.* **Operación Post para subir archivos**




- Visualización de archivos

*Fig 11.* **Operación Get para obtener metadatos**



### 3. Capa de Datos

Se describe la elección, el rol y la ubicación estratégica de la capa de datos en nuestra arquitectura. Se optó por utilizar PostgreSQL como motor de base de datos relacional para la gestión de usuarios, metadatos de los recursos y de logs generados por el sistema al momento de que un usuario sube un archivo.

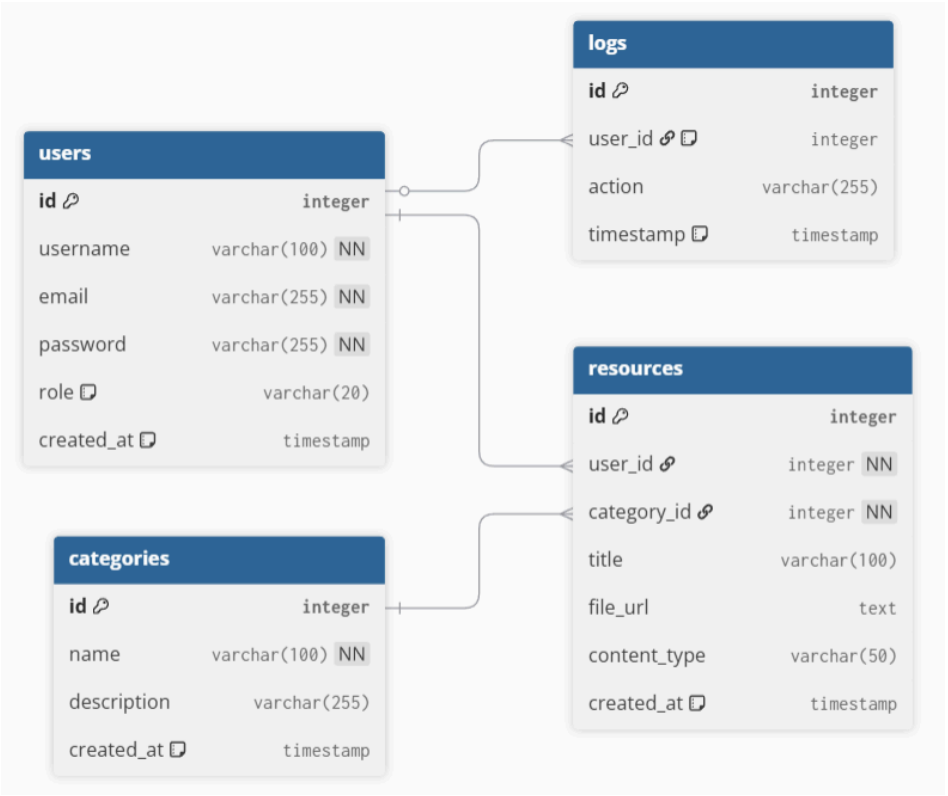
La base de datos se implementó como una instancia externa dedicada, separada de las instancias de la API. Con esto logramos una separación de responsabilidades, ya que no comparte recursos con las aplicaciones frontend y backend. La instancia se encuentra en la subred privada y el acceso está controlado por las reglas de Grupo de Seguridad, permitiendo conexiones sólo desde las instancias API por el puerto 5432.

*Fig 12*. **Configuración de la Instancia**


*Fig 13*. **Configuración de la Base de Datos y Usuario**


La base de datos cuenta con tres tablas para guardar usuarios, los recursos multimedia y la categoría de dichos de recursos. La tabla de recursos no almacena los archivos binarios, almacena la URL de ubicación del archivo dentro del Storage Service (Filestore).

*Fig 14*. **Modelo Relacional**




Las instancias de API se conectan a la base por medio de la red interna (Subred Privada) utilizando el puerto 5432 para realizar las operaciones de lectura y escritura.

*Fig 15.* **Conexión API a la base de datos**

```
// 1. CONEXIÓN A POSTGRESQL
const pool = new Pool({
  host: '10.7.112.3',
  user: 'app_user',
  password: '#Practica123',
  database: 'db-virtualizacion',
  port: 5432,
  ssl: { rejectUnauthorized: false }
});
```

**4. Integración con almacenamiento compartido**

Para lograr una integración con el almacenamiento compartido se usó la librería multer para utilizar una ruta absoluta ‘mnt/storage’ como destino de escritura, esta no es una ruta local, sino un punto de montaje que apunta la instancia de Filestore. A continuación se presentan imágenes que describen la configuración principal del almacenamiento compartido.

*Fig 16.* **Configuración de multer**

```
const storagePath = '/mnt/storage';
const storage = multer.diskStorage({
  destination: (req, file, cb) => cb(null, storagePath),
  filename: (req, file, cb) => cb(null, Date.now() + '-' + file.originalname)
});
const upload = multer({ storage: storage });
```

*Fig 17.* **Configuración de montaje persistente**

```
LABEL=cloudimg-rootfs   /        ext4   discard,commit=30,errors=remount-ro     0 1
LABEL=BOOT      /boot   ext4    defaults        0 2
LABEL=UEFI      /boot/efi       vfat    umask=0077      0 1
10.207.0.18:/archivos /mnt/storage nfs defaults 0 0
```

*Fig 18.* **Vinculación de la carpeta al servicio Filestore**

```
Filesystem             Size  Used Avail Use% Mounted on
/dev/root              8.7G  3.9G  4.8G  45% /
tmpfs                  2.0G     0  2.0G   0% /dev/shm
tmpfs                  783M  1.1M  782M   1% /run
tmpfs                  5.0M     0  5.0M   0% /run/lock
efivarfs                56K   24K   27K  48% /sys/firmware/efi/efivars
/dev/sda16             881M   65M  755M   8% /boot
/dev/sda15             105M  6.2M   99M   6% /boot/efi
10.207.0.18:/archivos 1007G  2.0M  956G   1% /mnt/storage
tmpfs                  392M   12K  392M   1% /run/user/1002
tmpfs                  392M   12K  392M   1% /run/user/1001
```

Finalmente, comprobaremos que el archivo subido en la sección anterior (Fig. 4) se encuentra en la carpeta ‘/mnt/storage’, para esto hacemos uso del comando ‘ls -l /mnt/storage’, donde podemos observar que el archivo ‘ExamenIA.html’ se encuentra dentro de la carpeta.

*Fig 19.* **Salida recortada del comando 'ls'**


Debido a que el balanceador de carga usa una plantilla de instancia para crear nuevas máquinas, todas tendrán la misma configuración, Con estas pruebas, hemos comprobado la integración con el servicio de almacenamiento compartido y la consistencia entre los nodos.

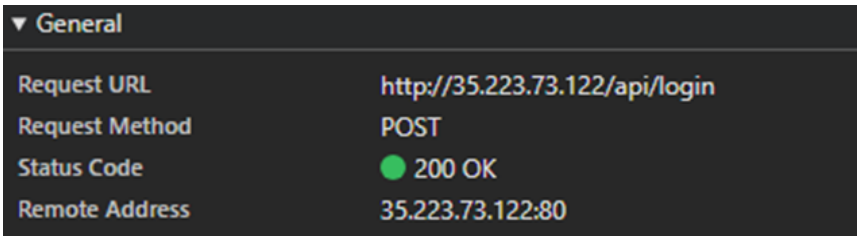
**5. Pruebas de conectividad integral**

Para concluir la parte 2, realizaremos 3 pruebas finales que comprobarán la funcionalidad y la integración de los servicios.

- La UI consume la API correctamente

Al pulsar el botón de iniciar sesión, podemos visualizar en la consola como la petición tiene un código 200, lo que significa que la interfaz se ha comunicado exitosamente con el servidor y ha recibido una respuesta.

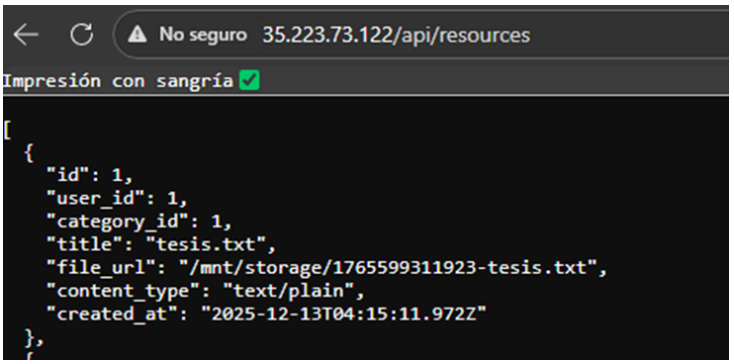
*Fig 20.* **Código 200 en la consola del navegador**




- La API accede a la base de datos sin problemas

El endpoint para obtener los datos de los archivos funciona correctamente, devolviendo un JSON con esta información.

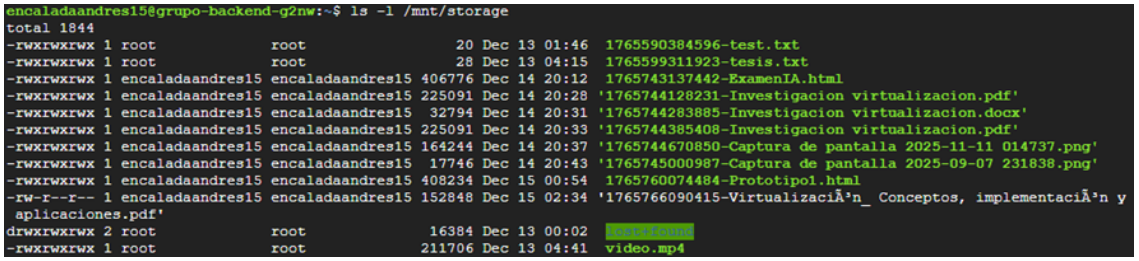
*Fig 21.* **Información obtenida por la API**




- El backend puede guardar y recuperar archivos desde el almacenamiento compartido

Al consultar la carpeta donde fue montado el servicio Filestore, observamos todos los archivos que han sido guardados en el servicio, los archivos 'tesis.txt' fueron archivos que se colocaron directamente para realizar pruebas, por lo que no presentan usuario.

*Fig 22.* **Salida del comando ls**



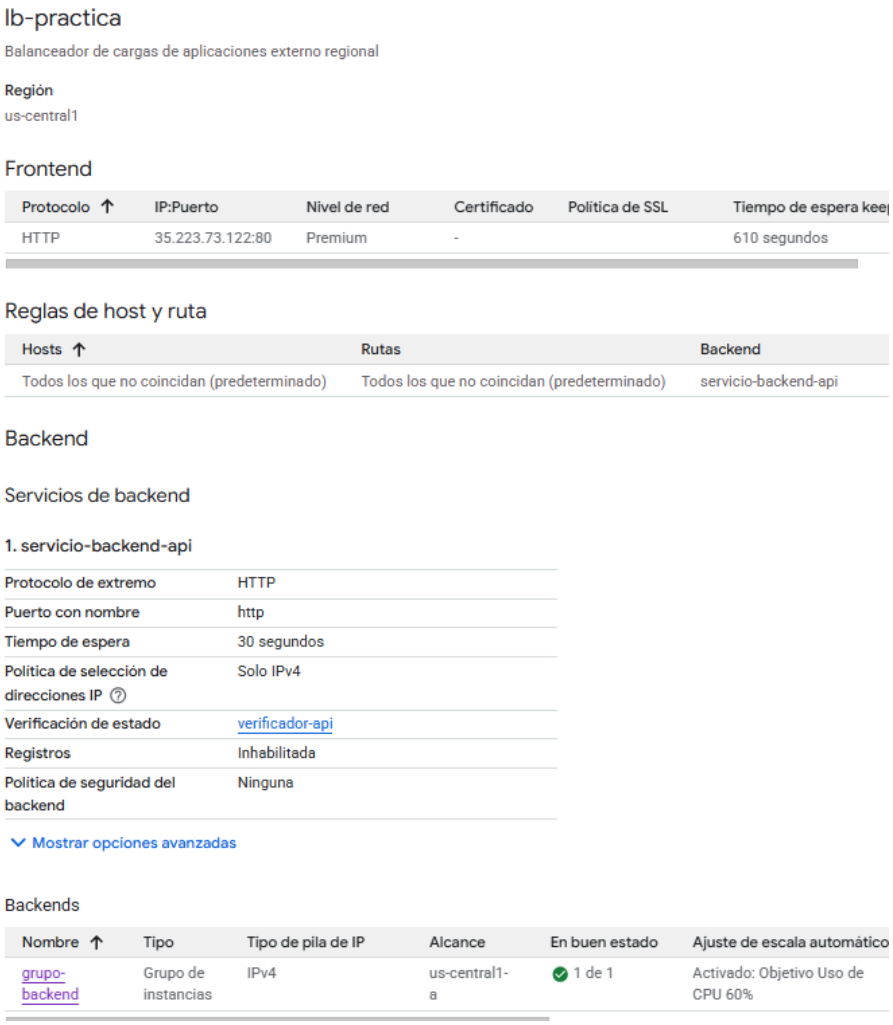

## Parte 3: Escalabilidad y balanceo

### 1. Balanceador de carga

En la siguiente Figura se muestra el balanceador de carga, junto con su IP y su puerto, además, podemos observar el grupo de instancias, el cual contiene una instancia (debido al bajo uso).

Se implementó un mecanismo de Health Check (Sondeo de estado) configurado en el protocolo TCP puerto 3000. Este servicio verifica periódicamente la disponibilidad de la API en cada instancia. Si una instancia falla o deja de responder, el balanceador la marca como ‘No saludable' y deja de enviarle tráfico automáticamente hasta que se recupere.

*Fig 23.* **Detalles del balanceador**



lb-practica

Balanceador de cargas de aplicaciones externo regional

Región

us-central1

Frontend

| Protocolo | IP:Puerto | Nivel de red | Certificado | Política de SSL | Tiempo de espera kee |
|---|---|---|---|---|---|
| HTTP | 35.223.73.122:80 | Premium | - | | 610 segundos |

Reglas de host y ruta

| Hosts | Rutas | Backend |
|---|---|---|
| Todos los que no coincidan (predeterminado) | Todos los que no coincidan (predeterminado) | servicio-backend-api |

Backend

Servicios de backend

1. servicio-backend-api

| | |
|---|---|
| Protocolo de extremo | HTTP |
| Puerto con nombre | http |
| Tiempo de espera | 30 segundos |
| Política de selección de direcciones IP | Solo IPv4 |
| Verificación de estado | verificador-api |
| Registros | Inhabilitada |
| Política de seguridad del backend | Ninguna |

Mostrar opciones avanzadas

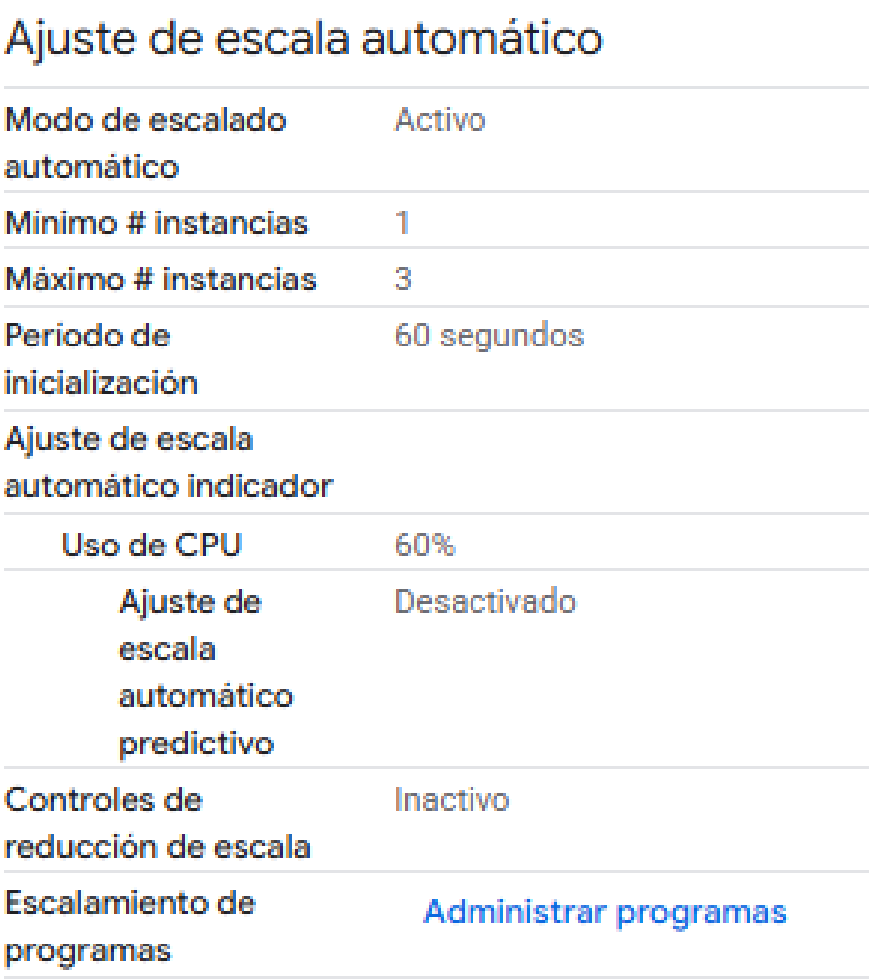
Backends

| Nombre | Tipo | Tipo de pila de IP | Alcance | En buen estado | Ajuste de escala automático |
|---|---|---|---|---|---|
| grupo-backend | Grupo de instancias | IPv4 | us-central1-a | 1 de 1 | Activado: Objetivo Uso de CPU 60% |

2. **Autoscaling**

Se configuró una política de autoescalado basada en métricas de rendimiento del sistema. Se definió un umbral de CPU del 60%: si el promedio de uso de CPU del grupo supera este valor, el orquestador aprovisiona nuevas instancias automáticamente. Se establecieron límites estrictos (Mín: 1, Máx: 3) para controlar el presupuesto y garantizar la disponibilidad mínima.

*Fig 24.* **Detalles de autoescalado**




- **Creación de componentes**

Para crear un balanceador de carga (donde se configura el autoescalado) debemos dirigirnos a la sección ‘Servicios de red’, donde encontraremos la pestaña balanceador de cargas, al crear un nuevo balanceador podremos definir distintos parámetros, como son el tipo (aplicaciones o red), orientación (público o uso interno), implementación (global o regional), se requerirán diferentes parámetros en función de la configuración elegida. A continuación, podremos definir varias configuraciones más, como frontends, backends y la red a la que pertenece el balanceador.

*Fig 25.* **Panel principal de la sección Servicios de red**

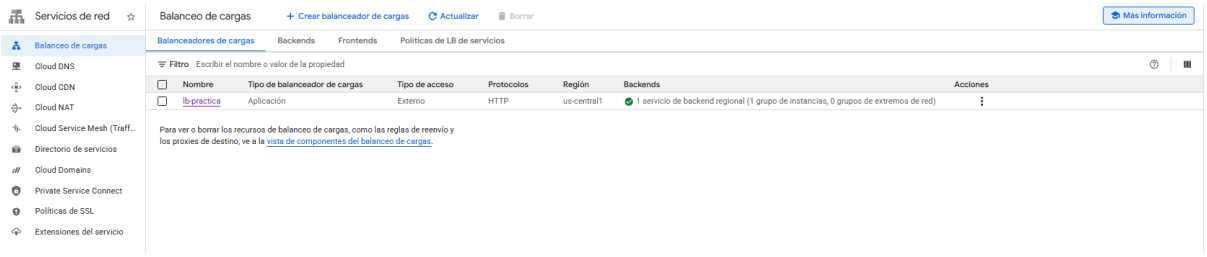
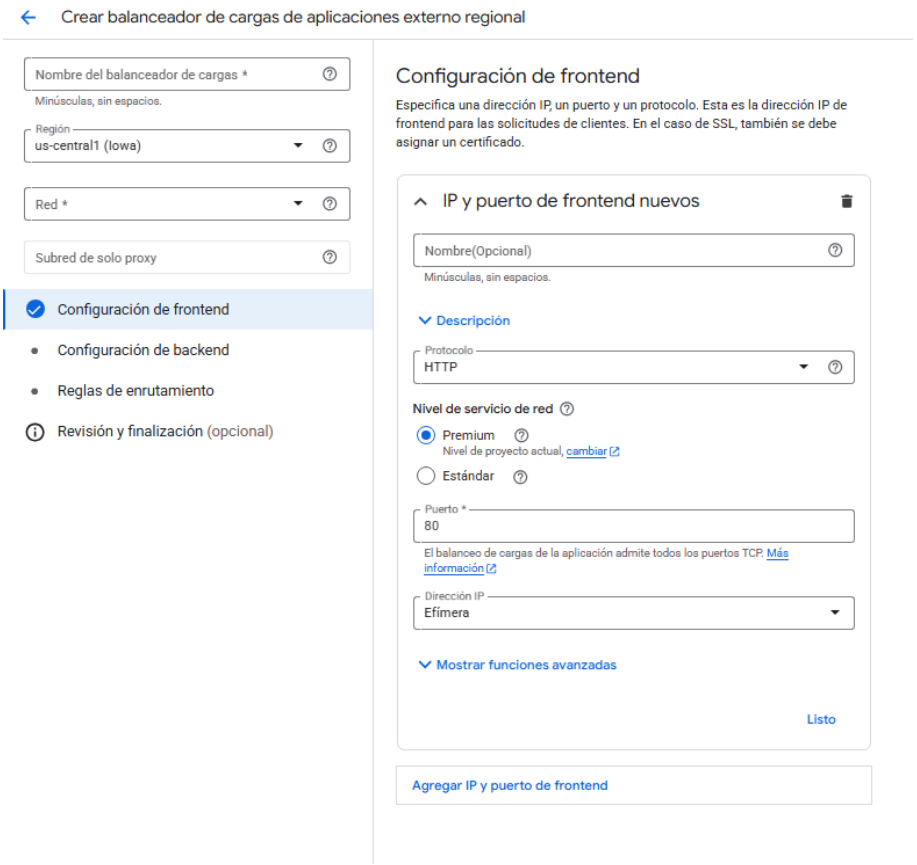


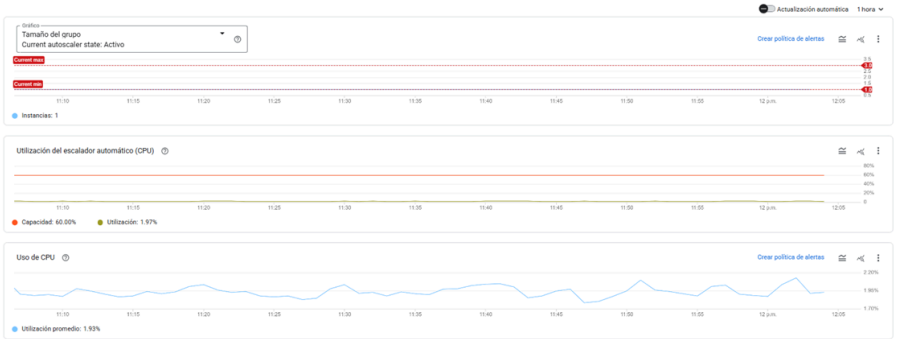

*Fig 26.* **Panel de configuración del balanceador de carga**




### 3. Pruebas de rendimiento

Finalmente, se realizaron pruebas para comprobar el funcionamiento del balanceador de carga. Primero, se observaron los datos del sistema cuando estaba en un estado tranquilo, los cuales pueden ser visualizados en el panel del grupo de instancias, estos datos pueden ser observados en la Fig. 15.

*Fig 27.* **Panel del grupo de instancias**



Posteriormente, se utilizó una herramienta llamada ‘stress-ng’ para saturar intencionalmente los núcleos de la instancia, logrando superar el umbral del 60%. A lo que el balanceador respondió creando más instancias, este cambio en el uso del CPU y el número de instancias puede ser visualizado en las siguientes figuras.

*Fig 28.* **Gráfica de uso del CPU**

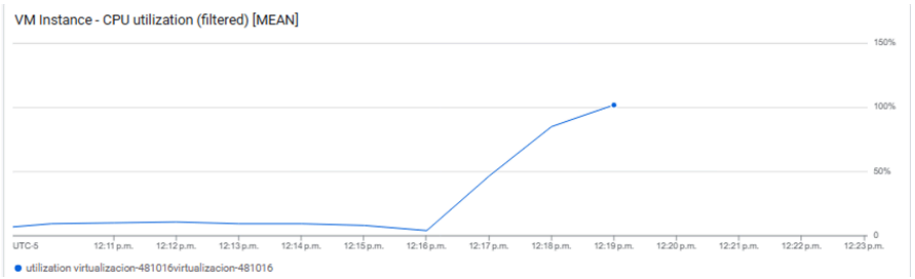
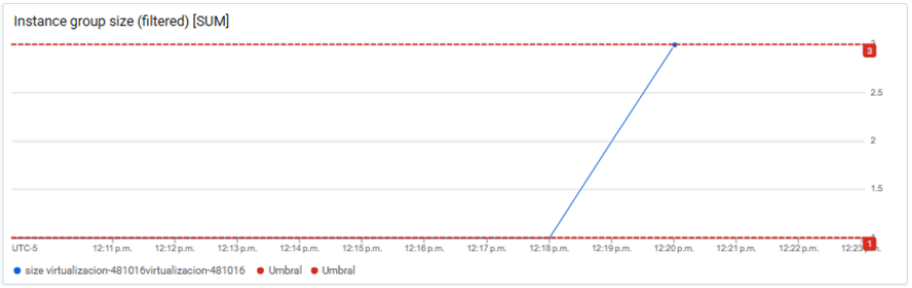



*Fig 29.* **Gráfica del número de instancias**




*Nota.* Debido a la simplicidad del proyecto se estableció un límite de solo 3 instancias.

*Fig 30.* **Panel de instancias del grupo de instancias**


Una vez el estresor es detenido, el uso del CPU baja y el balanceador comienza a eliminar instancias. Además, se observó un cambio en las gráficas del panel del balanceador, donde se observa un pico en la gráfica de uso del CPU.

*Fig 31.* **Detalles de instancias activas**

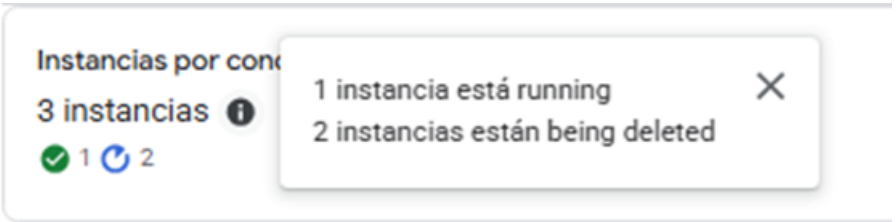


*Fig 32.* **Gráficas de estadísticas**

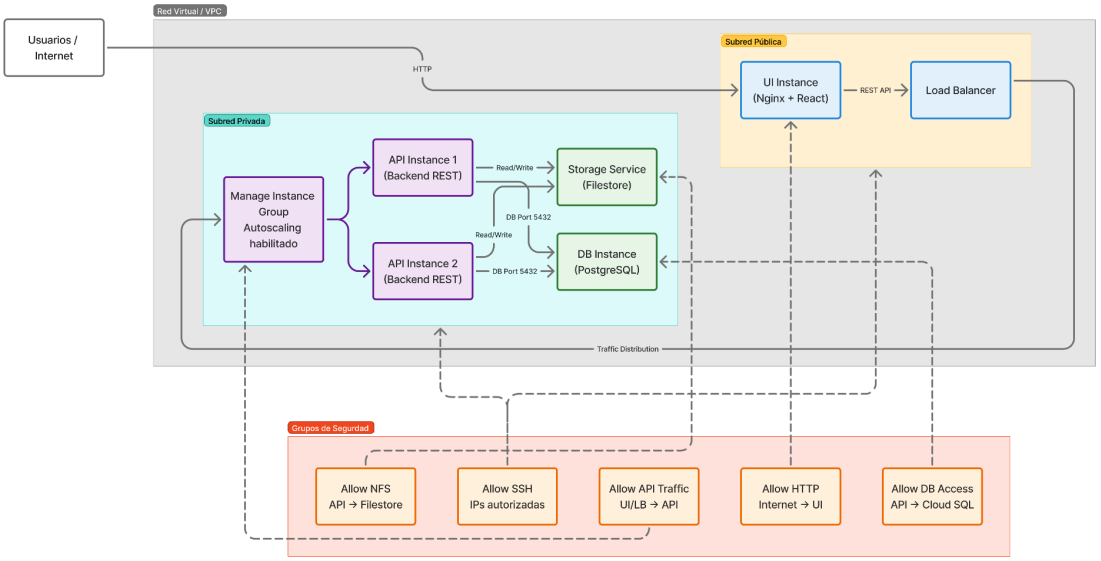

Finalmente, mediante PowerShell, se realizó una medida del tiempo de respuesta del sistema, donde se obtuvo un tiempo de 0.287604 segundos.

## Parte 4: Documentación

### 1. Diagrama Arquitectónico

En el siguiente diagrama se presenta la arquitectura del proyecto, resaltando elementos de redes, instancias, el balanceador de carga y el servicio de almacenamiento compartido.

*Fig 33.* **Diagrama arquitectónico**




### 2. Evidencias

Se creó un repositorio en github, donde se presentan los archivos principales del proyecto. Se separó el repositorio en carpetas para cada elemento (Backend y Frontend). Enlace al repositorio: https://github.com/AndresEncalada/Virtualizaci-n-en-la-Nube

## 3. Resultados obtenidos

Los resultados obtenidos demostraron que el Grupo de Instancias responde eficazmente ante picos de demanda, activando el autoescalado horizontal para duplicar la capacidad de cómputo  al superar el umbral de CPU del 60%, mientras que el Balanceador de Carga mantuvo la continuidad del servicio distribuyendo el tráfico sin interrupciones. Asimismo, se confirmó la consistencia integral de los datos, donde la sincronización entre Cloud SQL y el sistema de archivos compartido Filestore (NFS) aseguró que la información transaccional y los archivos multimedia fueran accesibles y persistentes desde cualquier instancia del clúster