

Universidad del Valle



Universidad del Valle Sede Tuluá

Proyecto 3: Despliegue de Servicio de Machine Learning en la Nube

Redes Neuronales

Msc. Carlos Andres Delgado Saavedra

Arango Guzman Juan Felipe - 202060066

Guerrero Jaramillo Carlos Eduardo - 202060216

Marulanda Valero John Jader - 202060034

Rivera Reyes Miguel Angel - 202059876

Tuluá - Valle del Cauca

2024

Tabla de Contenido.

1. Introducción.....	3
1.1. Objetivos del Proyecto.....	3
2. Desarrollo.....	4
2.1. Descripción del Entorno Local.....	4
2.2. Creación de Contenedores con Docker.....	6
2.3. Evidencias de Funcionamiento del Entorno Local.....	11
2.4. Documentación de Componentes del Proyecto.....	12
2.4.1. API Rest.....	12
1. Configuración del Entorno y Seguridad.....	12
2. Implementación del Modelo de Machine Learning.....	14
3. Endpoints.....	15
4. Dependencias.....	16
5. Flujo de Ejecución.....	16
2.4.2. Estructura de Comunicación - Consumo de endpoint.....	16
2.4.3. Vista Frontend.....	19
Estructura de Archivos.....	19
Funcionalidades Clave.....	20
Flujo de Trabajo.....	21
Código del Componente Central.....	21
3. Despliegue en la Nube.....	28
3.1. Proceso de Subida al Entorno Cloud.....	28
3.2. Recursos y Configuraciones del Entorno Cloud.....	30
3.3. Evidencias de Ejecución y Funcionalidad.....	31
4. Pruebas y Validación.....	32
4.1. Detalle de Pruebas en Entorno Local y Nube.....	32
5. Conclusiones.....	34
5.1. Resultados Obtenidos.....	34
5.2. Posibles Mejoras.....	35
6. Anexos.....	36
6.1. Código Fuente del Proyecto.....	36
6.2. Enlace video.....	36

1. Introducción

Este proyecto se centra en el despliegue de un servicio de Machine Learning en la nube, utilizando redes neuronales convolucionales. Se aborda desde su implementación local hasta su despliegue en entornos cloud, documentando cada paso para garantizar un flujo de trabajo reproducible y funcional. El propósito de este informe es detallar los objetivos, el alcance, y los pasos realizados para el desarrollo, implementación y validación del sistema.

1.1. Objetivos del Proyecto

El principal objetivo de este proyecto es desarrollar e implementar una solución funcional para el procesamiento de imágenes utilizando un modelo de red neuronal convolucional. Esto incluye:

- **Desarrollo de un backend:** Crear una API REST que interactúe con el modelo de Machine Learning para realizar predicciones.
- **Construcción de un frontend:** Diseñar una interfaz de usuario que facilite la interacción con el sistema, permitiendo la carga de imágenes y la visualización de resultados.
- **Despliegue en la nube:** Garantizar la accesibilidad y escalabilidad del servicio mediante el uso de plataformas cloud como Google Cloud Platform (GCP) o Microsoft Azure.
- **Validación del sistema:** Documentar pruebas locales y en la nube para demostrar el correcto funcionamiento y la experiencia del usuario.

1.2. Alcance

El alcance del proyecto incluye los siguientes aspectos:

Implementación local: Desarrollo de los componentes backend y frontend, asegurando su integración y correcto funcionamiento en un entorno controlado mediante Docker.

Despliegue en la nube: Configuración y puesta en marcha de los servicios en plataformas cloud, documentando cada paso para su replicabilidad.

Pruebas de funcionalidad: Evaluación del sistema mediante herramientas como Postman, Insomnia, tanto en el entorno local como en la nube.

Evidencias y documentación: Generación de un informe detallado con capturas de pantalla, logs y descripciones de cada proceso para sustentar los resultados obtenidos.

2. Desarrollo

2.1. Descripción del Entorno Local

Para la implementación local del proyecto, se configuró un entorno de desarrollo basado en tecnologías modernas tanto para el backend como para el frontend. La separación de componentes y el uso de contenedores aseguran un desarrollo organizado y reproducible.

Backend

El backend fue desarrollado utilizando **Python** y el framework **FastAPI**, conocido por su rendimiento y facilidad de uso para crear APIs RESTful. Se emplearon

diversas bibliotecas para manejar los modelos de Machine Learning y la interacción con el frontend:

- **fastapi (0.115.6):** Framework principal para la API.
- **uvicorn (0.34.0):** Servidor ASGI para ejecutar la aplicación FastAPI.
- **pydantic (2.10.4):** Validación de datos y definición de esquemas.
- **tensorflow:** Para la carga y ejecución del modelo de red neuronal convolucional.
- **sklearn:** Herramientas para preprocesamiento de datos y evaluación del modelo.
- **Pillow (PIL):** Manejo de imágenes para preprocesamiento y validación.

Adicionalmente, se configuraron dependencias específicas en un archivo requirements.txt para la instalación automatizada.

Frontend

El frontend fue desarrollado con **React** y **Vite**, utilizando **TypeScript** para garantizar tipado fuerte y escalabilidad del proyecto. Las principales tecnologías empleadas incluyen:

- **axios:** Para la comunicación con los endpoints del backend.
- **tailwindcss:** Para un diseño responsivo y moderno.
- **Vite:** Herramienta de construcción rápida para proyectos React.

Organización del Proyecto

Se crearon dos repositorios independientes para mantener la separación de responsabilidades:

1. **Repositorio Backend:** Contiene el código de la API, el modelo y las dependencias necesarias.
2. **Repositorio Frontend:** Incluye los componentes de la interfaz de usuario y el código que gestiona las interacciones con el backend.

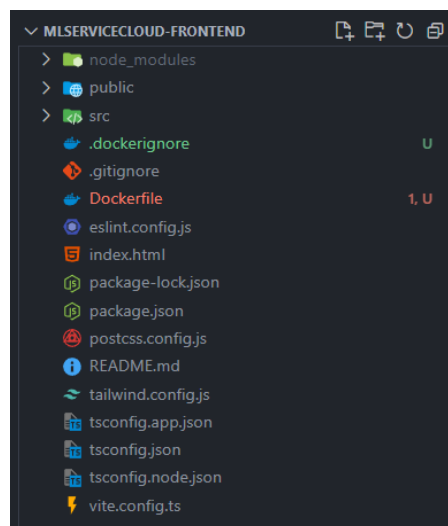
Entorno de Contenedores

Para garantizar la portabilidad, ambos componentes se integraron utilizando Docker. Se definieron archivos Dockerfile específicos para cada uno, Docker Compose para orquestar los dos contenedores.

2.2. Creación de Contenedores con Docker

Frontend Docker.

Para hacer una imagen Docker para nuestro Frontend, debemos crear primero los archivos Dockerfile para especificar la imagen que vamos a crear y .dockerignore para evitar que carpetas como node_modules o .vscode se copien al contenedor que vamos a crear.



Ahora, vamos a estructurar la imagen para crear el contenedor de nuestro frontend

```

1 # Construir el proyecto
2 FROM node:18 AS builder
3
4 # Configurar directorio de trabajo
5 WORKDIR /app
6
7 # Copiar archivos necesarios de las dependencias
8 COPY package.json package-lock.json ./
9
10 # Instalar dependencias
11 RUN npm install
12
13 # Copiar el resto de los archivos del proyecto
14 COPY . .
15
16 # Construir el proyecto
17 RUN npm run build
18
19 # Servidor web
20 FROM nginx:1.23-alpine
21
22 # Copiar el build generado anteriormente al servidor
23 COPY --from=builder /app/dist /usr/share/nginx/html
24
25 # Exponer el puerto 80
26 EXPOSE 80
27
28 # Comando de inicio del servidor
29 CMD ["nginx", "-g", "daemon off;"]

```

Asimismo, vamos a agregar las carpetas que no necesitamos en nuestro contenedor en el archivo .dockerignore

```

1 node_modules
2 dist
3 Dockerfile
4 .dockerignore
5 .vscode
6 .git

```

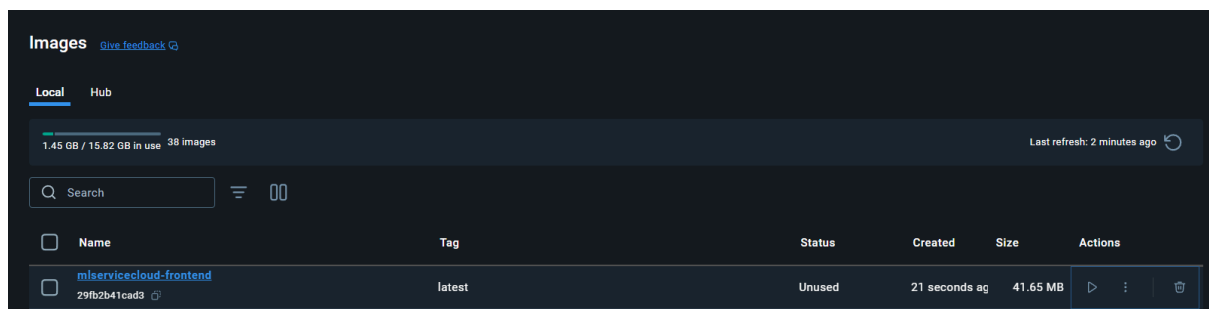
Ahora, ejecutamos el comando `docker build -t mlservicecloud-frontend .` con el fin de construir el contenedor

```

PS C:\Users\Carlos Guerrero\Documents\2024-2\Redes\MLServiceCloud-Frontend> docker build -t mlservicecloud-frontend .
[+] Building 6.9s (6/13)
=> extracting sha256:547a97583f72a32903ca1357d48fa302e91e8f83ffa18e0c40fd87adb5c06025 0.0s
=> extracting sha256:1f21f983520d9a440d410ea62eb0bda61a2b50dd79878071181b56b82efa9ef3 0.0s
=> extracting sha256:c23b4f8cf279507bb1dd3d6eb2d15ca84fac9eac215ab5b529aa8b5a060294c8 0.0s
=> [builder 1/6] FROM docker.io/library/node:18@sha256:b57ae84fe7880a23b389f8260d726b784010ed470c2ee26d4e2cbb955d25b12 3.9s
=> resolve docker.io/library/node:18@sha256:b57ae84fe7880a23b389f8260d726b784010ed470c2ee26d4e2cbb955d25b12 0.0s
=> sha256:b57ae84fe7880a23b389f8260d726b784010ed470c2ee26d4e2cbb955d25b12 6.41kB / 6.41kB 0.0s
=> sha256:9371b74049cb4bb6422c1f25c9f564ce9524a647fc951a35f0945efa3e96d4d 2.49kB / 2.49kB 0.0s
=> sha256:b2da2cb649e9f22d68ecc32c89d6833e6dc1705f0162c3f8a64df5772a478884 2.49kB / 2.49kB 0.0s
=> sha256:ce82e98d553dd62ca6a12bebf83992ae9f9ae2748275e74b66a68cc094f868b 35.65MB / 211.31MB 3.9s
=> sha256:6399a464889d3eae2913051cb98c35d0b6bfa20ec77d6b3a0461744a298a2a56 3.32kB / 3.32kB 1.6s
=> sha256:a3c94c84d15dfc1c2c202acca56d7327f541d62c10f9bc1dfb013a618aebd5f1 4.19MB / 45.70MB 3.9s
=> sha256:2cd8c50fd8ca9ed98f596af5d92400b4492b7b069d2d339a6ed8682fc568961 1.25MB / 1.25MB 3.7s
=> sha256:247468edfd9afcf43bf96caab52a1d979edd5eb13afcaf570c1513f4a35fa43f 0B / 446B 3.9s
[+] [internal] load build context 0.2s

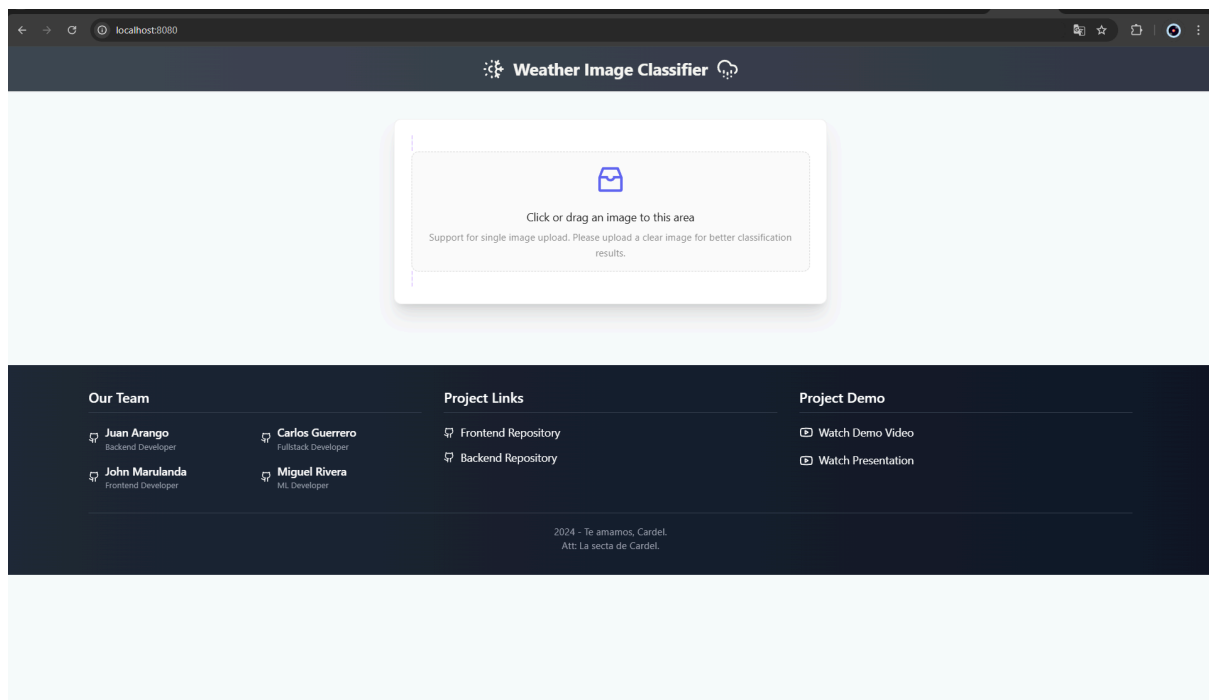
```

Verificamos que la imagen se construya con éxito



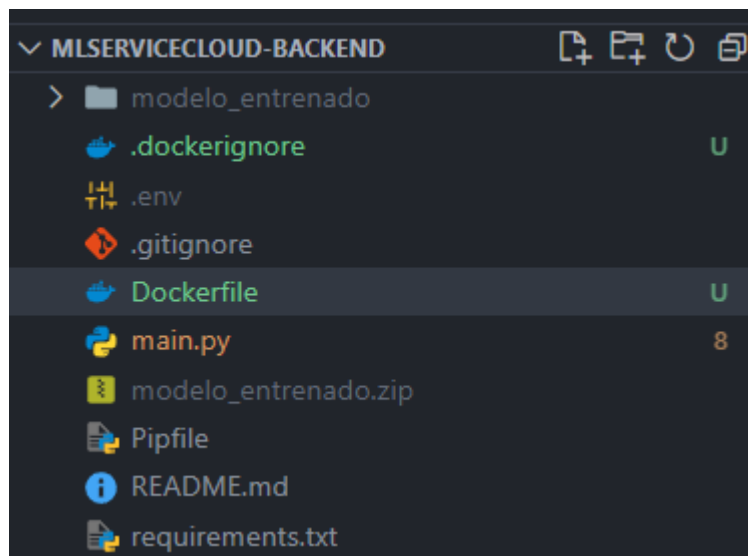
Finalmente, con el comando `docker run -p 8080:80 mlservicecloud-frontend` iniciamos la aplicación y exponemos el puerto 8080 para poder acceder desde un navegador

Verificamos que la aplicación cargue en nuestro navegador correctamente



Backend Docker

Para hacer una imagen Docker para nuestro Backend, debemos crear primero los archivos Dockerfile para especificar la imagen que vamos a crear y .dockerignore para evitar que carpetas como .vscode o .env se copien al contenedor que vamos a crear.



Ahora, vamos a estructurar la imagen para crear el contenedor de nuestro backend

```
Dockerfile > ...
1 C:\Users\Carlos Guerrero\Documents\2024-2\Redes\MLServiceCloud-Backend\Dockerfile •
2 Untracked
3
4 # Establecemos el directorio de trabajo
5 WORKDIR /app
6
7 # Copiamos el archivo de requerimientos
8 COPY requirements.txt .
9
10 # Instalamos las dependencias
11 RUN pip install --no-cache-dir -r requirements.txt
12
13 # Copiamos el resto de la aplicación al contenedor
14 COPY . .
15
16 # Exponemos el puerto para la aplicación
17 EXPOSE 8000
18
19 # Comando para ejecutar la aplicación FastAPI con Uvicorn
20 CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
21
```

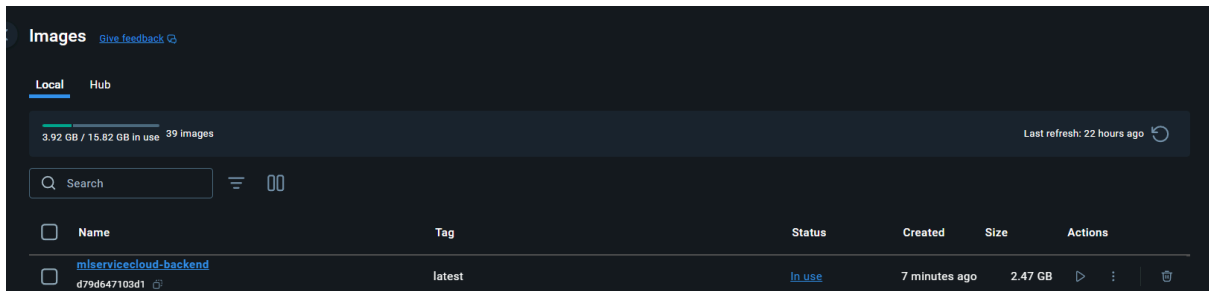
Asimismo, vamos a agregar las carpetas que no necesitamos en nuestro contenedor en el archivo .dockerignore

```
.dockerignore
1  .env
2  Pipfile
3  Pipfile.lock
4  README.md
5  modelo_entrenado.zip
6  __pycache__
7
```

Ahora, ejecutamos el comando `docker build -t mlservicecloud-backend .` con el fin de construir el contenedor

```
PS C:\Users\Carlos Guerrero\Documents\2024-2\Redes\MLServiceCloud-Backend> docker build -t mlservicecloud-backend .
[+] Building 20.5s (7/9)
-> extracting sha256:fd674058ff8f8cfa7fb8a20c006fc0128541cbbad7f7f7284f570d08f9e4d92 2.9s
-> extracting sha256:96df0e5e81799ba220e350fc342c1da017b41302df5954c705bee1407dcab03 0.3s
-> extracting sha256:75a2bc32319e3d6a5bc12888b074f86338a9e3e4304d99d1dddc07155b8ba76e 1.3s
-> extracting sha256:d381bf0bf6e4697fb23da0e7d50d080411236ffcd2853432e12fd0f6b372778 0.8s
-> [internal] load build context 6.9s
-> transferring context: 189.42MB 6.9s
-> [2/5] WORKDIR /app 0.4s
-> [3/5] COPY requirements.txt . 0.0s
-> [4/5] RUN pip install --no-cache-dir -r requirements.txt 10.6s
```

Verificamos que la imagen se creará con éxito



Name	Tag	Status	Created	Size	Actions
mlservicecloud-backend	latest	In use	7 minutes ago	2.47 GB	

Finalmente, con el comando `docker run -p -e API_KEY=<api_key> 8002:8000 mlservicecloud-backend` iniciamos la aplicación y exponemos el puerto 8002 para poder hacer peticiones.

2.3. Evidencias de Funcionamiento del Entorno Local

Como se evidencia en el punto anterior, podemos ver el funcionamiento en el entorno de docker.

Tanto en Frontend como en Backend, podemos evidenciar su funcionamiento en el entorno local:

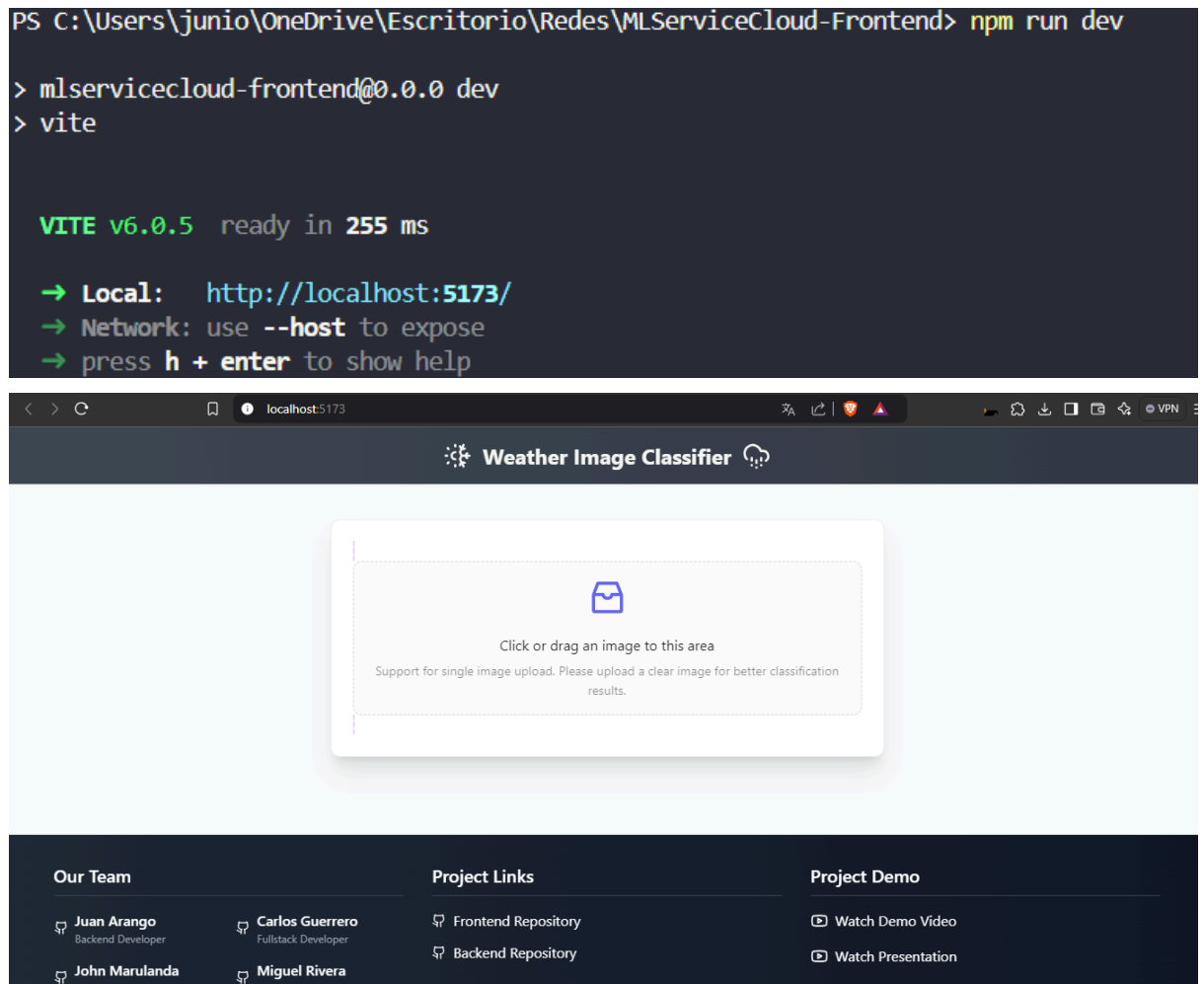
Backend:

```
(venv) PS C:\Users\junio\OneDrive\Escritorio\Redes\MLServiceCloud-Backend> uvicorn main:app --reload
INFO: Will watch for changes in these directories: ['C:\\Users\\junio\\OneDrive\\Escritorio\\Redes\\MLServiceCloud-Backend']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [8832] using StatReload

INFO: Started server process [14452]
INFO: Waiting for application startup.
INFO: Application startup complete.
█

INFO: 127.0.0.1 - "POST /predict/ HTTP/1.1" 200 OK
```

Frontend:



2.4. Documentación de Componentes del Proyecto

2.4.1. API Rest

Este backend está construido con **FastAPI**, un framework de Python conocido por su simplicidad y alto rendimiento. A continuación, desglosamos las partes principales:

1. Configuración del Entorno y Seguridad

- **Variables de Entorno:**

Python

```
from dotenv import load_dotenv
```

```
API_KEY = os.getenv("API_KEY")
```

Se utiliza dotenv para cargar la clave de acceso (API_KEY) desde un archivo .env. Esto garantiza que datos sensibles no se incluyan directamente en el código.

- **Middleware para Validar API Key:**

Python

```
@app.middleware("http")
async def api_key_validator(request: Request, call_next):
    api_key = request.headers.get("Authorization")
    if api_key != API_KEY:
        raise HTTPException(status_code=401, detail="API_KEY no válida o no
proporcionada")
    response = await call_next(request)
    return response
```

Este middleware intercepta cada solicitud para verificar si la clave API es válida, asegurando que solo usuarios autorizados puedan acceder a los endpoints.

- **CORS:**

Python

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
    expose_headers=["*"]
)
```

La configuración de CORS (Cross-Origin Resource Sharing) permite que esta API sea consumida desde diferentes dominios.

2. Implementación del Modelo de Machine Learning

- **Carga del Modelo:**

Python

```
model = tf.keras.models.load_model('./modelo_entrenado')
```

Se carga un modelo previamente entrenado y guardado en formato SavedModel.

- **Preprocesamiento de la Imagen:**

Python

```
img = Image.open(BytesIO(image_bytes)).convert("RGB")
img = img.resize((128, 128)) # Redimensionar la imagen
img_array = np.array(img, dtype=np.float32) / 255.0
img_array = tf.expand_dims(img_array, 0) # Expandir dimensiones para el batch
```

La imagen recibida se procesa para ajustarse al formato de entrada del modelo:

- Convertir a RGB.
- Redimensionar a 128x128.
- Normalizar los valores de píxeles (0-255 a 0-1).
- Expandir dimensiones para crear un batch de tamaño 1.

- **Predicción:**

Python

```
predictions = model.predict(img_array)
```

La imagen preprocesada se pasa al modelo para obtener predicciones.

- **Clasificación:**

Python

```
predicted_class = np.argmax(predictions[0], axis=-1)
```

- Se selecciona la clase con la probabilidad más alta. Además, se crea un diccionario que asocia índices con nombres de clases (rain, snow, etc.) para facilitar la interpretación de los resultados.

3. Endpoints

- **GET /:**

Python

```
@app.get("/")
async def home():
    return {"message": "Hello, World!"}
```

Endpoint básico que devuelve un mensaje para confirmar que el servidor está activo.

- **POST /predict/:**

Python

```
@app.post("/predict/")
async def predict(file: UploadFile = File(...)):
    ...
```

Este endpoint recibe una imagen como archivo, realiza predicciones utilizando el modelo y retorna un JSON con:

- La clase predicha (predicted_class_name).
- La probabilidad asociada a la clase (probability).
- Todas las probabilidades para cada clase (all_probabilities).

4. Dependencias

El archivo requirements.txt incluye todas las bibliotecas necesarias, como:

- tensorflow: Para cargar y ejecutar el modelo.
 - FastAPI y uvicorn: Para construir y ejecutar el servidor.
 - pillow: Para procesar imágenes.
 - numpy: Para manipulación de arrays.
 - python-dotenv: Para la gestión de variables de entorno.
-

5. Flujo de Ejecución

1. **Solicitud HTTP POST:**
 - El cliente envía una imagen junto con la clave API en el encabezado Authorization.
2. **Autenticación:**
 - El middleware verifica la API Key.
3. **Procesamiento de la Imagen:**
 - La imagen se redimensiona y se prepara para el modelo.
4. **Predicción:**
 - El modelo predice la clase meteorológica.
5. **Respuesta JSON:**
 - Se envía al cliente un JSON con los resultados..

2.4.2. Estructura de Comunicación - Consumo de endpoint

La comunicación entre el frontend y el backend sigue una arquitectura basada en **API REST**, asegurando un flujo claro y eficiente de datos. El backend, implementado con **FastAPI**, expone endpoints seguros y bien definidos, mientras que el frontend, desarrollado en **React** con **TypeScript**, los consume para ofrecer una experiencia interactiva al usuario.

Flujo de Comunicación

1. **Carga de la Imagen en el Frontend:**
 - El usuario selecciona o arrastra una imagen en el componente ImageUploader.tsx.
 - La imagen cargada se valida localmente para verificar que sea en formato PNG o JPG.

2. Preparación de la Solicitud:

- El frontend utiliza **axios** para crear una solicitud HTTP POST.
- La imagen se empaqueta en un objeto FormData para enviarla al backend como un archivo adjunto.

JavaScript

```
const formData = new FormData();
formData.append('file', fileList[0].originFileObj as Blob);
```

3. Autenticación con el API Key:

- La solicitud incluye un encabezado Authorization que contiene el API Key, requerido por el backend para validar el acceso.
- Esto asegura que solo usuarios autorizados puedan consumir el endpoint.

JavaScript

```
const response = await axios.post('/api/predict/', formData, {
  headers: {
    'Content-Type': 'multipart/form-data',
    'Authorization': API_KEY,
  },
});
```

4. Procesamiento en el Backend:

- El backend valida la solicitud, incluyendo la clave de API proporcionada.
- El archivo de imagen es procesado utilizando un modelo de aprendizaje profundo en TensorFlow.
- Se genera una predicción que incluye:
 - La clase más probable (predicted_class_name).
 - La probabilidad asociada a esa clase.
 - Todas las probabilidades para las clases disponibles.

5. Respuesta del Backend:

- El backend devuelve una respuesta JSON estructurada con los resultados de la clasificación.
- Ejemplo de respuesta:

Python

```
{
  "probability": 0.85,
  "predicted_class_name": "rain",
  "all_probabilities": {
    "rain": 0.85,
    "snow": 0.10,
    "fog": 0.05
  }
}
```

Visualización en el Frontend:

- El frontend procesa los datos recibidos del backend y los presenta al usuario en un cuadro de alerta con:
 - La clase predicha.
 - El porcentaje de confianza.
 - Una lista de probabilidades para todas las clases.

JavaScript

```
const { predicted_class_name, probability, all_probabilities } = response.data;
setResult({
  predicted_class_name,
  predicted_class_probability: probability,
  all_probabilities,
});
```

Flujo General de Comunicación

1. Frontend:

- Usuario selecciona una imagen.
- Envío de la imagen al backend mediante una solicitud POST.

2. Backend:

- Validación del API Key.
- Procesamiento del archivo de imagen.
- Generación de predicciones con el modelo de TensorFlow.
- Respuesta con los datos estructurados.

3. Frontend:

- Recepción de la respuesta.

- Presentación de resultados al usuario.

2.4.3. Vista Frontend

El frontend fue desarrollado con React utilizando TypeScript para garantizar un desarrollo más robusto y mantener un código más estructurado y escalable. Además, Vite fue utilizado como herramienta de construcción para reducir tiempos de recarga durante el desarrollo, mejorando así la productividad.

Se integraron herramientas clave como:

- **Ant Design:** Biblioteca de componentes para crear interfaces modernas y estilizadas.
 - **Axios:** Para realizar solicitudes HTTP al backend.
 - **TailwindCSS:** Estilización de componentes.
-

Estructura de Archivos

El código está organizado en carpetas y archivos clave, lo que facilita la navegación y el mantenimiento del proyecto:

1. **components/:**

- **Header.tsx:** Componente que muestra el encabezado principal con el título del proyecto.
- **ImageUploader.tsx:** Componente central para la carga y clasificación de imágenes.
- **Footer.tsx:** Proporciona información del equipo y enlaces útiles.

2. **pages/:**

- **MainPage.tsx:** Combina los componentes en una interfaz funcional, sirviendo como la página principal.
-

Funcionalidades Clave

1. **Carga de Imágenes:**

- Utiliza el componente Dragger de **Ant Design** para implementar una zona interactiva de "arrastrar y soltar".
- Admite la carga de una sola imagen en formatos PNG y JPG.

2. **Previsualización:**

- Una vez cargada la imagen, se genera una previsualización para verificar que sea correcta antes de enviarla.

3. **Clasificación de Imágenes:**

- Al presionar el botón "**Classify Image**", la imagen se envía al backend mediante una solicitud HTTP POST usando **axios**.
- La clave API se incluye en los encabezados para autenticación.

4. **Visualización de Resultados:**

- Los resultados se presentan en un cuadro de alerta. Incluyen:
 - La clase predicha.
 - La probabilidad asociada.
 - Todas las probabilidades ordenadas por clase.

5. **Gestión de Errores:**

- Si ocurre un error, como un formato de archivo no válido o problemas de conexión con el backend, se muestra un mensaje de alerta claro para el usuario.

Flujo de Trabajo

1. El usuario ingresa a la interfaz principal.
 2. Sube una imagen utilizando la zona de "arrastrar y soltar" o seleccionándola desde su sistema de archivos.
 3. Visualiza una previsualización de la imagen.
 4. Presiona el botón "Classify Image" para enviarla al backend.
 5. Obtiene los resultados de la clasificación presentados de forma clara.
 6. Si lo desea, puede reiniciar el proceso subiendo otra imagen.
-

Código del Componente Central

1. Configuración del Componente:

JavaScript

```
const [fileList, setFileList] = useState<UploadFile[]>([]);

const [loading, setLoading] = useState(false);

const [result, setResult] = useState<PredictionResult |
null>(null);

const [error, setError] = useState<string | null>(null);
```

2. Variables de estado para manejar:

- La lista de archivos cargados.
- El estado de carga durante la clasificación.
- Los resultados obtenidos del backend.
- Mensajes de error.

3. Carga y Previsualización de Imágenes:

JavaScript

```
const uploadProps: UploadProps = {

  beforeUpload: (file) => {

    const isImage = file.type === 'image/png' || file.type ===
    'image/jpeg';

    if (!isImage) {

      setError('Only PNG and JPG image files are allowed!');

      return Upload.LIST_IGNORE;

    }

    return false; // Permitir la carga si el formato es correcto

  },

  fileList,

  onChange: ({ fileList: newFileList }) => {

    setFileList(newFileList);
```

```
setError(null); // Limpia errores previos

setResult(null); // Limpia resultados previos

},

maxCount: 1,

};
```

4. Validaciones incluidas:

- Permitir solo imágenes en formato PNG y JPG.
- Limitar la carga a un solo archivo.

5. **Clasificación de Imágenes:**

JavaScript

```
const handleUpload = async () => {

  if (fileList.length === 0) return;

  setLoading(true);

  setError(null);

  setResult(null);
```

```
const formData = new FormData();

formData.append('file', fileList[0].originFileObj as Blob);

try {

    const response = await axios.post('/api/predict/', formData,
    {

        headers: {

            'Content-Type': 'multipart/form-data',

            'Authorization': API_KEY,

        },

    });

    const { predicted_class_name, probability,
    all_probabilities } = response.data;

    setResult({ predicted_class_name,
    predicted_class_probability: probability, all_probabilities });

} catch (err: any) {
```



```
        setError(err.response?.data?.detail || 'Error processing the  
        image. Please try again.');
```

```
    } finally {  
  
        setLoading(false);  
  
    }  
  
};
```

6. Este código se encarga de:

- Enviar la imagen al backend.
- Manejar los errores de conexión o validación.
- Actualizar los resultados con las predicciones.

Capturas de Pantalla

Se incluye una muestra de la interfaz de usuario, así como el flujo típico desde la carga de una imagen hasta la visualización de resultados.

⚙️ Weather Image Classifier 🖱️




Click or drag an image to this area

Support for single image upload. Please upload a clear image for better classification results.

Our Team


 **Juan Arango**
Backend Developer

 **John Marulanda**
Frontend Developer

 **Carlos Guerrero**
Fullstack Developer


 **Miguel Rivera**
ML Developer


Project Links

 [Frontend Repository](#)

 [Backend Repository](#)

Project Demo

 [Watch Demo Video](#)

 [Watch Presentation](#)




Click or drag an image to this area

Support for single image upload. Please upload a clear image for better classification results.



Click or drag an image to this area

Support for single image upload. Please upload a clear image for better classification results.

 [Copy](#)



Click or drag an image to this area

Support for single image upload. Please upload a clear image for better classification results.

 GettyImages-695548638-1024x668.jpg



Classify Image

⏪ Classify Image



Processing your image...



Classify Image



Classification Result

Upload Another

Predicted Class: sandstorm

Confidence: 96.56%

All Probabilities:

rain: 0.04%
snow: 0.86%
sandstorm: 96.56%
rime: 0.04%
dew: 0.28%
fogsmog: 0.62%
frost: 0.02%
hail: 0.20%
glaze: 0.30%
rainbow: 0.82%
lightning: 0.27%

3. Despliegue en la Nube

3.1. Proceso de Subida al Entorno Cloud

Backend

Para configurar el entorno en Google Cloud, debemos habilitar el Artifact Registry, para subir nuestro contenedor Docker con el modelo

cloud

ImgClassifierProject

artifact registry

Buscar

registry

Crear repositorio

Nombre *

imgclassifierrepository

Formato

☒ Docker

☐ Maven

☐ npm

☐ Python

☐ Apt

☐ Yum

☐ Canalizaciones de Kubeflow

☐ Go

☐ Genérico [VISTA PREVIA](#)

Modo

☒ Estándar

☐ Remoto

☐ Virtual

Tipo de ubicación

☒ Región

☐ Multirregión


Región *

us-east1 (Carolina del Sur)

Google Cloud

ImgClassifierProject

Detalles del producto



Artifact Registry API

Google Enterprise API

HABILITAR

PROBAR ESTA API [↗](#)

DESCRIPCIÓN GENERAL

PRECIOS

DOCUMENTACIÓN

PRODUCTOS RELACIONADOS

Descripción general

With Artifact Registry you can store and manage your build artifacts (e.g. Docker images, Maven packages, npm packages), in a scalable and integrated repository service built on Google infrastructure. You can manage repository access with IAM and interact with repositories via gcloud, Cloud Console, and native package format tools. The service can also be integrated with Cloud Build and other CI/CD systems. Artifact Registry abstracts away infrastructure management, so you can focus on what matters most – delivering value to the users of your services and applications. Note: Enabling the Artifact Registry API will not affect your use of Container Registry in the same project.

[Más información ↗](#)

Detalles adicionales

Tipo: [SaaS & APIs](#)

Última actualización del producto: 30/4/22

Categoría: [DevOps](#), [Google Enterprise APIs](#)

Nombre del servicio: artifactregistry.googleapis.com

```
Terminal - charles@charles-vivobook:~  
[charles@charles-vivobook ~]$ docker tag mlservicecloud-backend:latest us-east1-docker.pkg.dev/imgclassifierproject/imgclassifierepository/model  
[charles@charles-vivobook ~]$ docker push us-east1-docker.pkg.dev/imgclassifierproject/imgclassifierepository/model  
Using default tag: latest  
The push refers to repository [us-east1-docker.pkg.dev/imgclassifierproject/imgclassifierepository/model]  
880a538bacae: Pushed  
25fcac302c54: Pushed 26fea57c5558a78a290ccff9bfa430674b1cd17839e7b5d5146d49102  
6913db0f31ce: Pushed  
bf5b220d06cc: Pushed  
c9b39419f641: Pushed  
211bd86efa05: Pushed  
7cc4fe6ec513: Pushed  
8b296f486960: Pushed  
latest: digest: sha256:df6499d26fea57c5558a78a290ccff9bfa430674b1cd17839e7b5d5146d49102 size: 1999  
[charles@charles-vivobook ~]$
```

3.2. Recursos y Configuraciones del Entorno Cloud

Comienza tu prueba gratuita con un crédito de \$300. No te preocupes, no se te cobrará si se acaban los créditos. [Más información](#)

IGNORAR [COMENZAR GRATIS](#)

Google Cloud

ImgClassifierProject

Buscar (/) recursos, documentos, productos y más

Buscar

Artifact Registry

Imágenes de imgclassifierepository

BORRAR

EDITA EL REPOSITORIO

INSTRUCCIONES DE CONFIGURACIÓN

ACTUALIZAR

Repositorios

Configuración

us-east1-docker.pkg.dev

imgclassifierepository

imgclassifierepository

Detalles del repositorio

Formato Docker

Tipo Estándar

MOSTRAR MÁS

Filtro

Ingresar el nombre o el valor de la propiedad

<input type="checkbox"/> Nombre ↑	Conexión	Fecha de creación	Actualizado
<input type="checkbox"/> model	—	hace 2 minutos	hace 2 minutos

Notas de versión

41

Google Cloud

ImgClassifierProject

Buscar (/) recursos, documentos, productos y más

Buscar

Cloud Run

Crear servicio

MOSTRAR LÍNEA DE COMANDOS

Cada servicio expone un extremo único y ajusta automáticamente la escala de la infraestructura subyacente para controlar las solicitudes entrantes. No se puede cambiar el nombre del servicio ni la región más adelante.

Artifact Registry

Implementar una revisión desde una imagen de contenedor

Docker Hub

Implementar continuamente a partir de un repositorio (de origen o de función)

GitHub

Usar un editor directo para crear una función [VISTA PREVIA](#)

Funciones

Usar un editor directo para crear una función [VISTA PREVIA](#)

URL de la imagen del contenedor

us-east1-docker.pkg.dev/imgclassifierepository/model:latest

SELECCIONAR

REALIZAR PRUEBAS CON UN CONTENEDOR DE MUESTRA

Debe detectar las solicitudes HTTP en SPOR y no depender del estado local. [Cómo se configura un contenedor?](#)

Configurar

Nombre del servicio *

model

Para usar esta opción, debe estar habilitada la API de Cloud Run Admin.

HABILITAR

URL del extremo

https://model-7746048f9923.region.run.app

Autenticación *

☐ Permitir invocaciones sin autenticar

Marca esta opción si estás creando una API pública o un sitio web.

CREAR

CANCELAR

Resumen de precios

Precios de Cloud Run

Nivel gratuito

Primeras 180,000 unidades de CPU virtuales segundo por mes

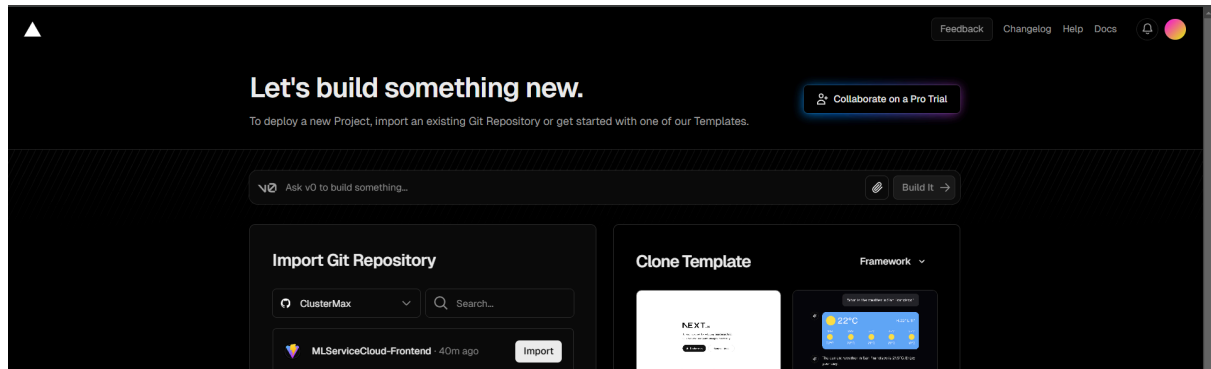
Primeros 360,000 GiB segundo por mes

2 millones de solicitudes por mes

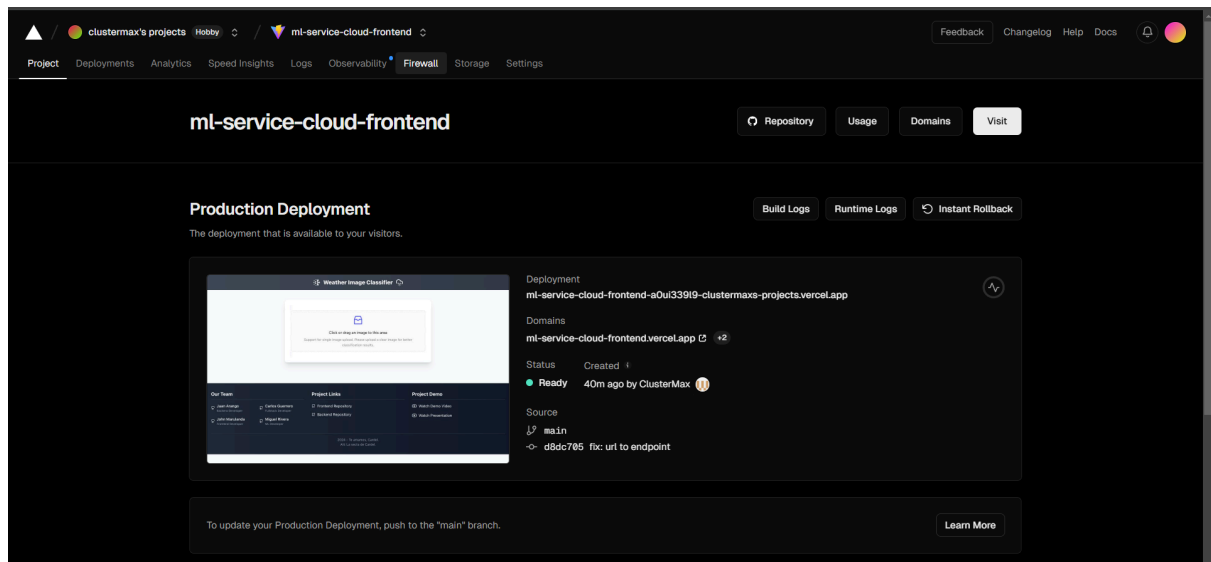
Verificar los detalles de los niveles pagados

Frontend

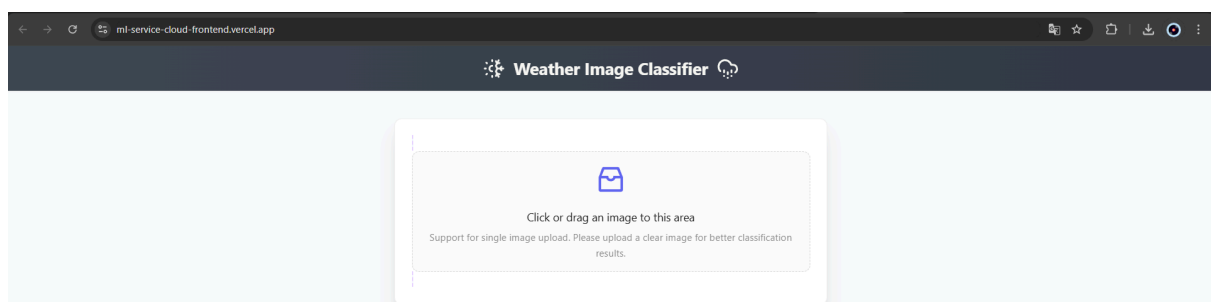
En Vercel, importamos el proyecto desde el repositorio GitHub



Nos creará una instancia donde estará alojada la aplicación

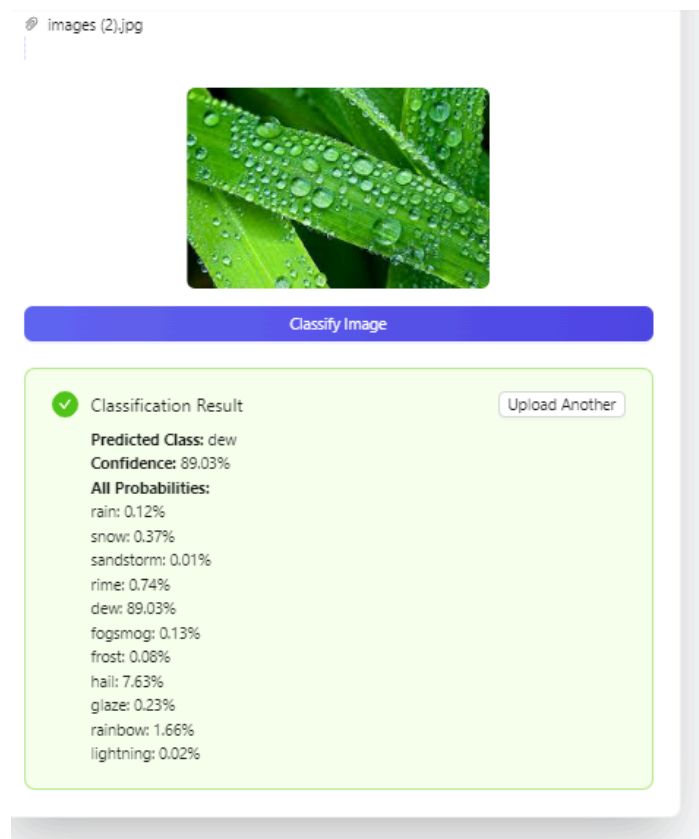
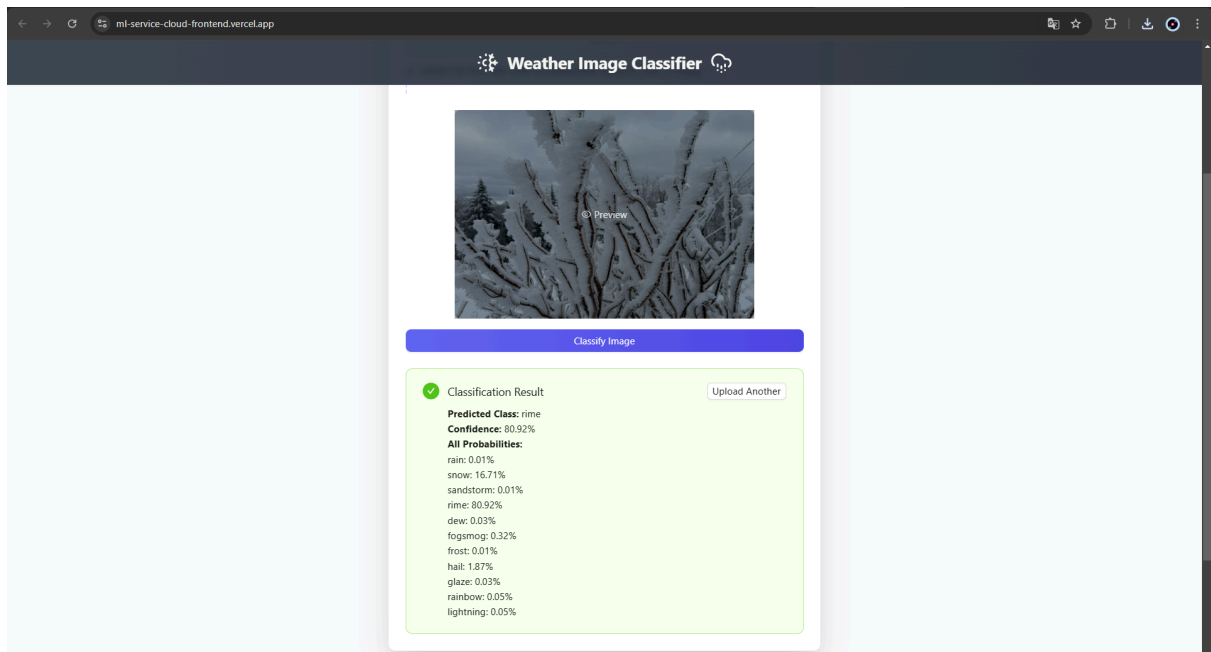


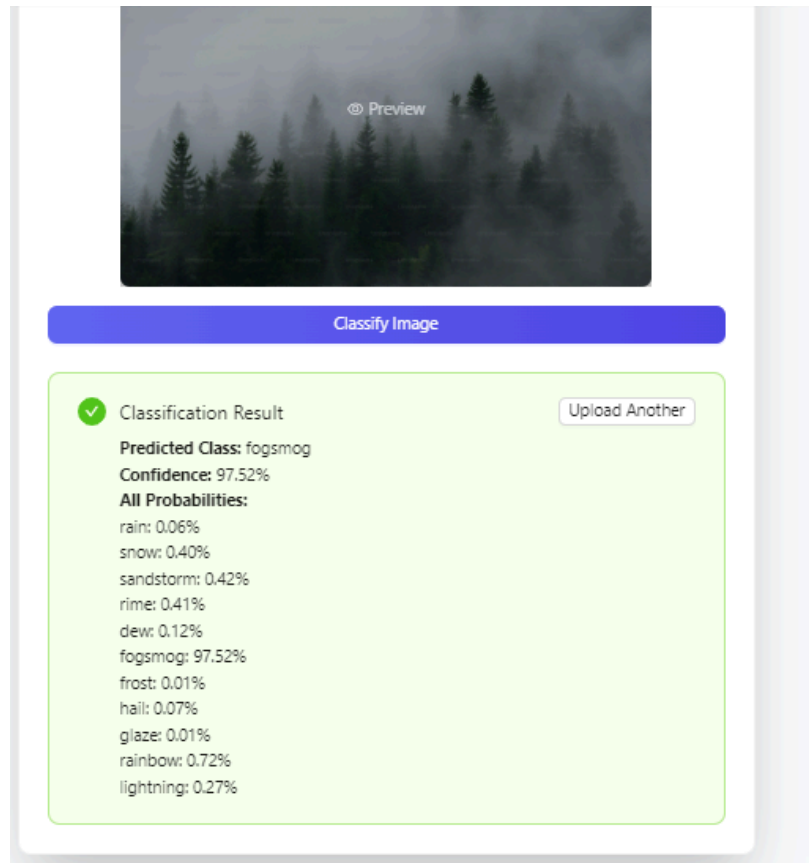
Finalmente, podemos acceder desde la url que nos proporciona Vercel



3.3. Evidencias de Ejecución y Funcionalidad

Una vez configurado todo, podemos hacer la prueba desde Vercel, la cuál está conectada a la implementación del modelo en Google Cloud



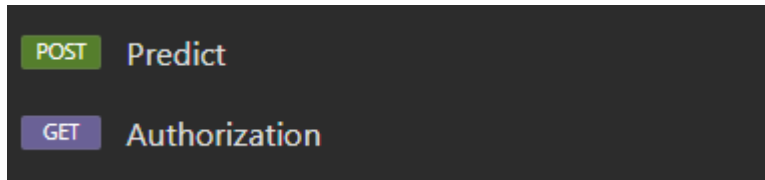


4. Pruebas y Validación

Para la validación y prueba de los endpoints desarrollados, se utilizaron herramientas como Postman e Insomnia, que facilitaron la construcción y verificación de solicitudes HTTP hacia el backend. Las URLs de consultas incluyeron tanto la dirección local, **<http://localhost:8000/api/predict/>**, como la versión en la nube, **<https://imgclassifier-774604876933.us-east1.run.app/predict/>**. Ejemplos de respuestas obtenidas mediante estas herramientas ilustraron el correcto funcionamiento del sistema: en ambos entornos se recibieron datos en formato JSON con las predicciones generadas, permitiendo corroborar que el modelo de clasificación operaba de manera consistente y confiable.

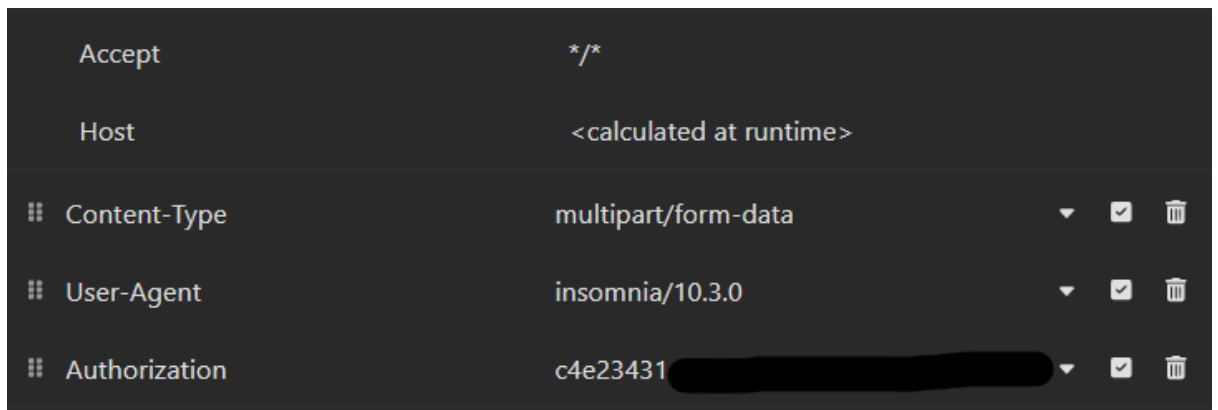
4.1. Detalle de Pruebas en Entorno Local y Nube.

En este caso hicimos uso de Insomnia, donde creamos dos peticiones HTTP

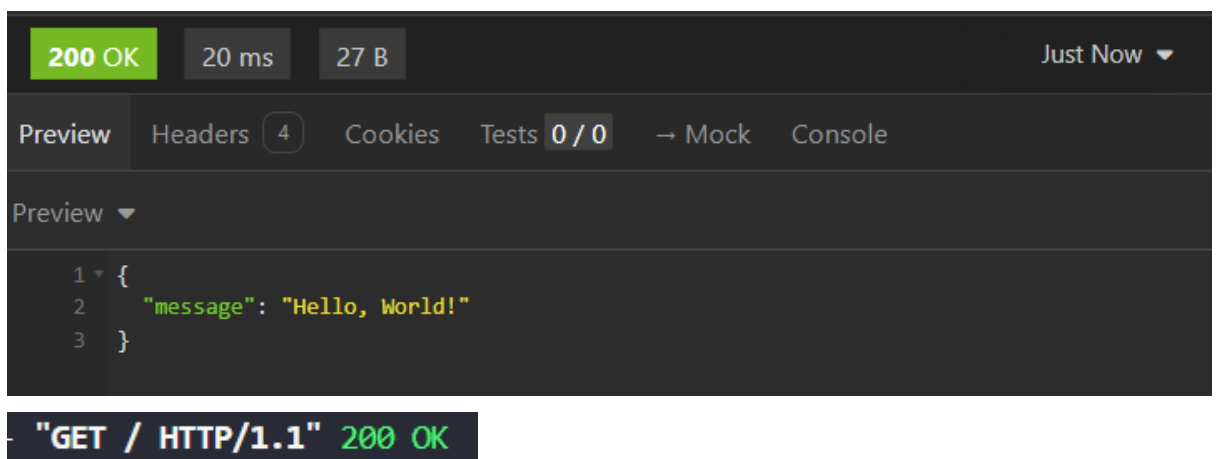


La de autorización es para confirmar el API KEY con una respuesta del servidor:

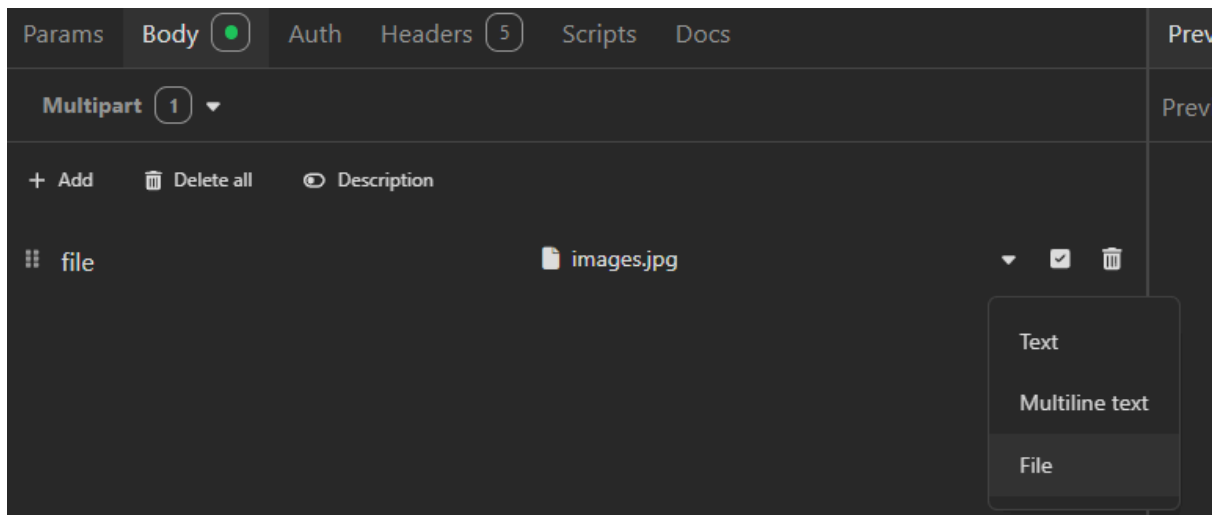
Se organizaron los HEADERS de ambas peticiones:



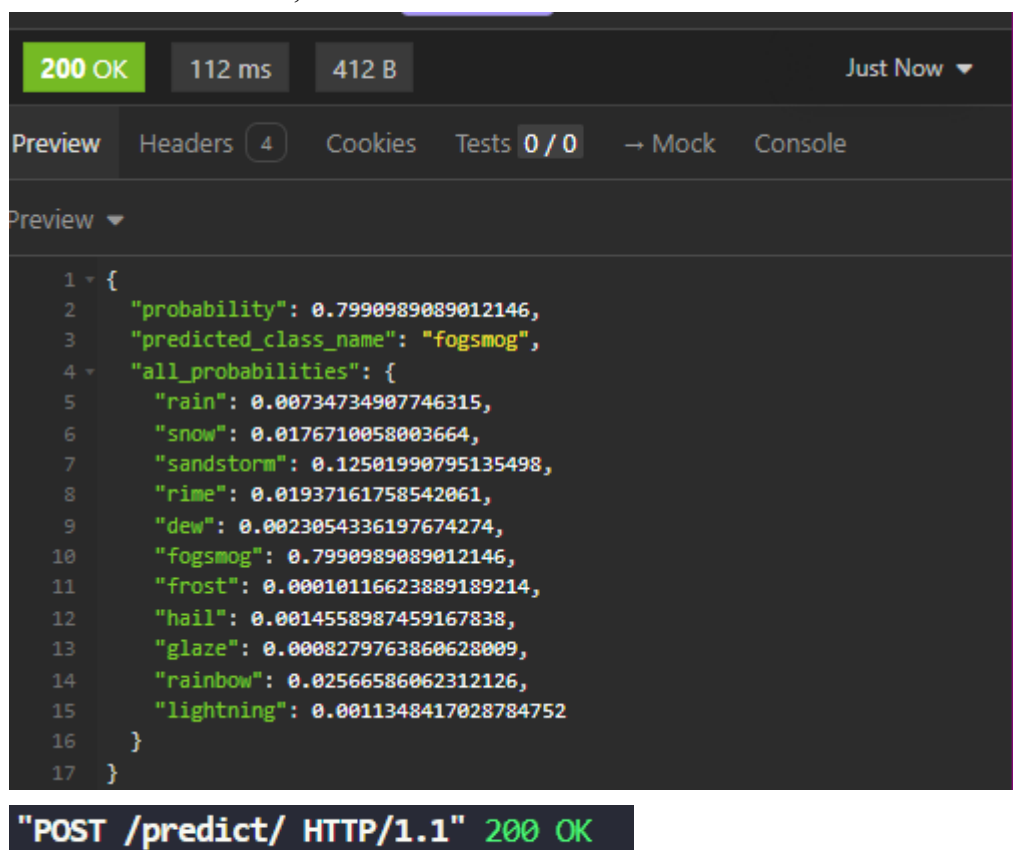
Al enviar la petición GET y si todo esta correcto, recibimos un mensaje de Hello World:



Cuando enviamos la petición POST, se configuran los headers de la misma manera, y enviamos una imagen:



Y si todo está correcto, obtenemos la salida:



5. Conclusiones

5.1. Resultados Obtenidos

El proyecto logró cumplir con los objetivos planteados inicialmente, destacándose los siguientes logros:

1. Desarrollo e Integración Exitosa:

- Se desarrolló un backend funcional utilizando FastAPI, capaz de procesar imágenes y realizar predicciones mediante un modelo de redes neuronales convolucionales.
- El frontend, construido con React y TypeScript, ofrece una interfaz de usuario intuitiva que facilita la carga de imágenes y la visualización de resultados.

2. Portabilidad y Escalabilidad:

- Gracias al uso de contenedores Docker, se logró garantizar la portabilidad del sistema, asegurando su correcto funcionamiento tanto en el entorno local como en la nube.
- El despliegue en la nube fue exitoso, permitiendo el acceso al sistema desde cualquier ubicación mediante plataformas como Vercel para el frontend y Google Cloud para el backend.

3. Validación Consistente:

- Las pruebas realizadas, tanto en el entorno local como en la nube, confirmaron la precisión y fiabilidad del sistema. Se obtuvieron resultados consistentes en ambas configuraciones, con respuestas rápidas y correctas del modelo.

5.2. Posibles Mejoras

A pesar de los resultados positivos, se identificaron áreas que podrían ser mejoradas en futuras iteraciones:

1. Optimización del Modelo:

- Evaluar modelos más avanzados o técnicas de optimización para reducir los tiempos de inferencia y mejorar la precisión en casos complejos.

2. Gestión de Errores:

- Ampliar el manejo de errores tanto en el backend como en el frontend para proporcionar mensajes más específicos y guiar mejor al usuario ante problemas como formatos de archivo no soportados o fallos en la conectividad.

3. Escalabilidad del Backend:

- Implementar balanceadores de carga o soluciones de escalado automático en el entorno cloud para manejar un mayor volumen de solicitudes simultáneas.

4. Mejoras en la Experiencia de Usuario:

- Incorporar elementos visuales adicionales, como gráficos o indicadores en tiempo real, para enriquecer la presentación de los resultados.

5. Pruebas Extendidas:

- Realizar pruebas más exhaustivas con diferentes conjuntos de datos y condiciones para evaluar el desempeño del sistema en escenarios diversos.

6. Anexos

6.1. Código Fuente del Proyecto

- Frontend: <https://github.com/JohnMarulanda/MLServiceCloud-Frontend>
- Backend: <https://github.com/JohnMarulanda/MLServiceCloud-Backend>

6.2. Enlace video

Enlace: <https://youtu.be/lvb8Gw3CL24>