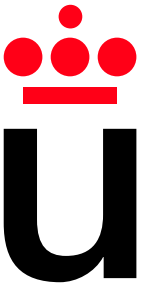


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

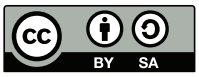
Grado en Ingeniería Informática

Curso 2025-2026

Trabajo Fin de Grado

DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA PARA LA ORQUESTACIÓN DE MODELOS DE IA EN ENTORNOS CLOUD

Autor: Manuel Gonzalo Ramírez Tirado
Tutor: Michel Maes Bermejo



Agradecimientos

A mis amigos de la universidad, por acompañarme durante estos años y crecer juntos hasta convertirnos en informáticos.

A mi hermano, mi familia y mis amigos del colegio, por no parar nunca de apoyarme y estar a mi lado siempre que los he necesitado.

A mi tutor, Michel, por su amabilidad, apoyo y ayuda para sacar adelante este proyecto.

Finalmente, quiero dar las gracias a mi madre. Poder contar siempre con ella, ahora y en lo que me depare el futuro, es lo que me ha empujado a lograr todo lo que he conseguido. Gracias, eres una inspiración.

Resumen

Este Trabajo Fin de Grado presenta el diseño e implementación de una arquitectura en la nube orientada a la gestión y orquestación eficiente de modelos de inteligencia artificial (IA). Toda la solución se ha desplegado íntegramente en el ecosistema de Amazon Web Services (AWS), aprovechando su amplio conjunto de servicios para garantizar escalabilidad, mantenibilidad y seguridad.

Con el objetivo de validar la arquitectura propuesta, se ha desarrollado un caso de uso práctico centrado en la restauración de imágenes, en el que se integran distintos modelos de IA capaces de realizar tareas de eliminación de ruido, aumento de resolución y eliminación de texto superpuesto (*inpainting*). Cada modelo se ha desplegado como un microservicio independiente desarrollado en FastAPI, mediante imágenes de Docker y orquestado a través de Amazon ECS. Además, se ha automatizado el proceso de despliegue utilizando Terraform bajo el enfoque de *Infraestructura como Código* (IaC).

Los resultados obtenidos demuestran la viabilidad y robustez de la arquitectura propuesta, que destaca por su modularidad, tolerancia a fallos y capacidad de adaptación. Además, la implementación confirma el potencial de AWS como entorno completo y flexible para el despliegue de aplicaciones basadas en inteligencia artificial. Como líneas futuras de trabajo, se plantea, entre otras, el uso de servicios con GPU, como Amazon SageMaker o Amazon EC2, la mejora de la capa de persistencia de datos con Amazon RDS y la ampliación de los mecanismos de autenticación, para una gestión más segura de los usuarios.

Palabras clave:

- Vue.js
- Python
- Terraform
- Infrastructura como Código (IaC)
- FastAPI
- Docker
- Amazon Web Services

- Amazon CLoudFront
- Amazon ECR
- Amazon ECS
- Amazon S3

Abstract

This Final Degree Project presents the design and implementation of a cloud-based architecture aimed at the efficient management and orchestration of artificial intelligence (AI) models. The entire solution has been deployed within the Amazon Web Services (AWS) ecosystem, taking advantage of its broad range of cloud services to ensure scalability, maintainability, and security.

To validate the proposed architecture, a practical use case has been developed focused on image restoration, integrating several AI models capable of performing tasks such as image denoising, super-resolution, and text removal (inpainting). Each model is deployed as an independent microservice using FastAPI, containerized with Docker, and orchestrated through Amazon ECS. The infrastructure and deployment processes have been automated through Terraform, following an Infrastructure as Code (IaC) approach.

The results shows the viability of the proposed architecture, which achieves high modularity, fault tolerance, and adaptability to different workloads. Moreover, the implementation confirms the potential of AWS as a complete and flexible environment for deploying AI-based applications. Future work may focus on improving performance using GPU-based services such as Amazon SageMaker or Amazon EC2, enhancing the data persistence layer with Amazon RDS, and extending the authentication mechanisms to offer a more secure user management system.

Keywords:

- Vue.js
- Python
- Terraform
- Infrastructure as Code (IaC)
- FastAPI
- Docker
- Amazon Web Services
- Amazon CLoudFront

- Amazon ECR
- Amazon ECS
- Amazon S3

Índice de contenidos

Índice de tablas

Índice de figuras

Índice de códigos

1

Introducción y Motivación

La inteligencia artificial (IA) constituye una de las disciplinas tecnológicas más influyentes, decisivas e incluso revolucionarias de la última década. Sus aplicaciones abarcan desde la visión artificial (computer vision) y el procesamiento del lenguaje natural, hasta la robótica y la toma de decisiones automatizadas. Algunas de ellas se han popularizado y son de uso frecuente, como los asistentes virtuales, las herramientas de creación de contenido y búsqueda, los asistentes de escritura, los sistemas de análisis de datos orientados a predicciones y a otras muchas aplicaciones, etc. El carácter innovador, dinámico y en constante evolución de esta tecnología, la convierte en un área de investigación y desarrollo de gran relevancia en el ámbito académico, empresarial y social.

De forma paralela, la computación en la nube o cloud computing desempeña, cada vez más, un papel esencial en la transformación digital. Las tecnologías en la nube ofrecen la capacidad de acceder a recursos computacionales bajo demanda a través de Internet, eliminando así la necesidad de contar con infraestructuras físicas propias y reduciendo significativamente los costes de mantenimiento y escalabilidad, a la vez que facilitan el aumento de la flexibilidad, la productividad y la seguridad. Gracias a esta tecnología, organizaciones de todos los tamaños pueden aprovechar las infraestructuras disponibles y distribuidas, para desplegar, ejecutar y mantener aplicaciones, de manera flexible y eficiente. Como consecuencia, pueden enfocarse fundamentalmente en su negocio y no en las infraestructuras que necesiten.

La convergencia entre la inteligencia artificial y las tecnologías cloud representa una oportunidad estratégica llena de posibilidades: la nube no solo permite entrenar y desplegar modelos de IA de última generación, sino que también faci-

lita su integración en sistemas de producción de forma escalable y accesible. Para desarrollar y optimizar de un modo eficiente todas las posibilidades y oportunidades de estas tecnologías, es imprescindible diseñar arquitecturas que simplifiquen y optimicen el ciclo de vida de los modelos de IA: desde su orquestación, gestión y automatización, hasta su consumo por parte de las aplicaciones y usuarios finales.

En este Trabajo Fin de Grado (TFG) se propone el diseño e implementación de la arquitectura de una aplicación orientada a la gestión eficiente de modelos de inteligencia artificial, con especial énfasis en la facilidad de despliegue, la escalabilidad y la mantenibilidad. Tanto la aplicación como los modelos de IA que la integran se despliegan íntegramente en el ecosistema cloud de Amazon Web Services (AWS), aprovechando su amplio catálogo de servicios para garantizar un funcionamiento robusto, seguro y adaptable a distintos escenarios de uso. Con el fin de validar la arquitectura propuesta, se ha desarrollado un caso de uso práctico que consiste en el despliegue de tres modelos de inteligencia artificial especializados en tareas de restauración de imágenes. Las funcionalidades de estos modelos son el aumento de la resolución, la reducción de ruido y la eliminación de texto superpuesto de las imágenes.

2

Objetivos

El **objetivo principal** de este trabajo es diseñar, implementar y desplegar una arquitectura cloud que permita gestionar de manera paralela, modular y escalable modelos de inteligencia artificial. La propuesta se centra en aprovechar los servicios de Amazon Web Services (AWS), para ofrecer propuestas que garanticen sin ninguna excepción: facilidad de despliegue, alta disponibilidad, seguridad y mantenibilidad en la gestión de múltiples modelos de IA. Para poner a prueba esta arquitectura, se implementará un caso de uso donde se desplegarán tres modelos de inteligencia artificial enfocados a la restauración de imágenes (eliminación de ruido, aumento de resolución y eliminación de texto superpuesto).

Para alcanzar este objetivo principal, se ha definido una serie de objetivos específicos que articulan el desarrollo del proyecto:

1. **Diseño de la arquitectura cloud**:
    - Diseñar una arquitectura modular que permita la incorporación de nuevos modelos de inteligencia artificial sin afectar al funcionamiento del sistema.
    - Identificar y seleccionar los servicios de Amazon Web Services (AWS) necesarios para implementar la arquitectura (Amazon API Gateway, Amazon ECS Fargate, Amazon DynamoDB, Amazon S3, Amazon WAF, Amazon CloudFront..., entre otros).
    - Establecer la relación entre estos servicios, asegurando siempre una integración válida, coherente y eficiente.

2. **Infraestructura como código**:
    - Definir toda la infraestructura del proyecto mediante Terraform, aplicando el enfoque de infraestructura como código (IaC).
    - Asegurar la reproducibilidad del código, así como la facilidad para escalar o modificar la infraestructura en el futuro.
3. **Frontend**:
    - Diseñar una interfaz gráfica intuitiva y accesible, que permita al usuario interactuar con los modelos de inteligencia artificial y le posibilite evaluar la eficacia de la infraestructura desplegada.
    - Desplegar la interfaz de la aplicación utilizando Amazon CloudFront, con el fin de reducir latencias, optimizar su rendimiento y garantizar una distribución global eficiente.
4. **Backend**:
    - Desarrollar, mediante FastAPI, diferentes microservicios independientes para cada modelo de inteligencia artificial, asegurando: modularidad, separación de responsabilidades y alto rendimiento en las peticiones.
    - Diseñar tres modelos de inteligencia artificial enfocados en la restauración de imágenes: Un modelo de aumento de resolución, un modelo de eliminación de ruido y un modelo de eliminación de texto superpuesto en la imagen.
    - Incorporar mecanismos de balanceo de carga a través de Amazon Network Load Balancer (NLB) y Amazon ECS Service Connect, optimizando la distribución del tráfico y la resiliencia del sistema.
5. **Despliegue de contenedores**:
    - Desarrollar e implementar los microservicios en contenedores Docker, garantizando la portabilidad y facilitando su despliegue en Amazon ECS Fargate.
    - Gestionar las imágenes de contenedores en Amazon ECR, facilitando el versionado y la actualización de los modelos de IA.
6. **Seguridad y autenticación**:
    - Incorporar un sistema de autenticación de usuarios mediante Amazon Cognito.
    - Implementar políticas de seguridad y control de acceso, utilizando Amazon WAF y aplicando reglas específicas para mitigar cualquier ataque común.

7. **Almacenamiento y persistencia de datos**:
   - Utilizar Amazon DynamoDB, para almacenar y consultar la información de las APIs de modelos de IA.
   - Integrar Amazon S3 como repositorio para el almacenamiento de imágenes procesadas por los modelos de inteligencia artificial.
   - Configurar Amazon CloudWatch, para el monitoreo de métricas del sistema y registro de logs.
8. **Automatización de procesos**:
   - Desarrollar funciones Lambda para tareas auxiliares, como, por ejemplo, la eliminación periódica de las imágenes almacenadas.
   - Integrar Amazon EventBridge, como mecanismo de orquestación de eventos que disparen flujos automáticos dentro del sistema.
9. **Validación y pruebas**:
   - Realizar pruebas de integración, para verificar la comunicación entre los distintos servicios de AWS.
   - Ejecutar pruebas de carga, para evaluar el rendimiento de la arquitectura frente a múltiples peticiones simultáneas.
   - Realizar pruebas de integración y pruebas unitarias, para garantizar el buen funcionamiento, tanto de la aplicación como de las APIs.

3

Tecnologías, Herramientas y Metodologías

A continuación, se presentan de manera detallada las **tecnologías y herramientas** (secciones 3.1 y 3.2) que se han empleado para el desarrollo, despliegue y puesta en funcionamiento de la aplicación. Así mismo, se describen las **metodologías** (sección 3.3) que se han utilizado durante el proceso de implementación, para asegurar una construcción ordenada y mantenible de la aplicación, alineada permanentemente con todas las buenas prácticas de la ingeniería de software.

3.1. Tecnologías

Las tecnologías seleccionadas abarcan distintos ámbitos de la solución: el **frontend**, que se encarga de la interacción con el usuario; el **backend**, responsable de la lógica de la aplicación y la orquestación de los modelos de inteligencia artificial; y la **infraestructura**, que garantiza la escalabilidad, disponibilidad y despliegue automático en la nube. En las siguientes subsecciones se detalla cada una de ellas, junto con el porqué y la justificación de su elección, así como las ventajas que nos han aportado para desarrollar el proyecto.

3.1.1. Frontend

JavaScript

JavaScript [1] es el lenguaje de programación que se ha empleado para desarrollar el frontend. Este lenguaje permite dotar a las páginas de interactividad y flexibilidad, gestionar eventos y realizar peticiones asíncronas al servidor.

Una de las ventajas de JavaScript es su versatilidad: puede ejecutarse tanto en el navegador (frontend) como en el servidor (backend, mediante entornos como Node.js). En este proyecto, ha sido utilizado de manera indirecta a través de Vue.js, que aprovecha sus capacidades para simplificar el desarrollo de interfaces reactivas y modulares.

Vue.js

Para crear el frontend, se ha utilizado **Vue.js** [2], un framework progresivo de JavaScript que posibilita desarrollar interfaces de usuario de manera rápida y modular. Se ha elegido Vue.js, por su simplicidad y por la posibilidad de crear componentes reutilizables. Esto facilita el mantenimiento y la escalabilidad de la aplicación.

3.1.2. Backend

Python

En el desarrollo del backend, se ha empleado **Python** [3], un lenguaje de programación versátil y con una amplia comunidad. Python destaca por su legibilidad, gran cantidad de librerías disponibles y su popularidad en el ámbito de la inteligencia artificial y el procesamiento de datos, lo que lo convierte en una elección natural y obvia para este proyecto.

FastAPI

Para la construcción de las APIs, se ha utilizado **FastAPI** [4], un framework moderno de Python diseñado para crear APIs rápidas y eficientes. FastAPI destaca por su rendimiento, basado en **Starlette** y **Pydantic**, y por su capacidad para generar de forma automática la documentación de la API (Swagger y Redoc). Gracias a su enfoque asíncrono, se ha podido implementar endpoints con tiempos de respuesta reducidos. Esta cualidad resulta especialmente importante y se ha tenido muy en cuenta en nuestro caso de uso, ya que, en el caso de imáge-

nes, los modelos de inteligencia artificial requieren un tiempo de procesamiento bastante alto.

3.1.3. Librerías de Inteligencia Artificial

PyTorch

PyTorch [5] es una de las librerías de deep learning más utilizadas en la actualidad. Proporciona un entorno flexible y eficiente para la creación, entrenamiento y despliegue de redes neuronales, gracias a su integración con un amplio entorno de herramientas. En este proyecto, se ha empleado PyTorch como base para implementar y ejecutar los modelos de inteligencia artificial.

DeepInv

DeepInv, presentada pòr J. Tachella et al. en [6], es una librería desarrollada para la investigación y aplicación de métodos de deep learning en problemas inversos, como la deconvolución, el desenfoque o la eliminación de ruido en imágenes. Su modularidad permite diseñar arquitecturas variacionales y redes neuronales para la restauración de imágenes. En este proyecto, se ha empleado específicamente para tareas de **denoising**, y con ella se ha logrado la reducción de ruido en imágenes de entrada.

Real-ESRGAN NCNN

Real-ESRGAN NCNN, presentada por X. Wang et al. en [7], es una implementación optimizada del modelo Real-ESRGAN (Enhanced Super-Resolution Generative Adversarial Network). Permite realizar **upscaling** de imágenes de manera eficiente, mediante el uso de la librería NCNN, orientada a dispositivos de alto rendimiento. Esta herramienta nos ha permitido mejorar la resolución de las imágenes manteniendo la coherencia visual y reducir artefactos.

PaddleOCR

PaddleOCR, creada por B. Lai, B. Yu et al. en [8], es una librería basada en el marco de trabajo **PaddlePaddle**. Ha sido diseñada para la detección y reconocimiento óptico de caracteres en imágenes. Ofrece modelos preentrenados, capaces de identificar texto en múltiples idiomas y contextos. En este proyecto, se ha utilizado como apoyo en las tareas de **inpainting**, lo que ha facilitado la localización y reconstrucción de las regiones que contenían texto en las imágenes.

Lama Cleaner

Lama Cleaner, desarrollada por R. Suvorov, E. Logacheva et al. en [9], es una aplicación y librería que implementa el modelo LaMa (Large Mask Inpainting), especializado en la tarea de **inpainting**. Posibilita eliminar o rellenar regiones de una imagen de manera automática y realista, aprovechando técnicas avanzadas de visión artificial. En el marco de este proyecto, se ha utilizado Lama Cleaner para la edición y restauración de imágenes, eliminando elementos no deseados y corrigiendo zonas dañadas.

3.1.4. Infraestructura

Terraform

En el ámbito de la infraestructura, se ha utilizado **Terraform** [10], una tecnología de **Infrastructura como Código (IaC)**[11], que permite definir y desplegar los recursos en la nube de forma declarativa.

La elección de Terraform nos ha posibilitado versionar la infraestructura en nuestro repositorio y aplicar cambios de manera controlada. En nuestro caso, se ha empleado para desplegar recursos en AWS, tales como instancias de las APIs, el frontend de la app, y servicios gestionados y configuraciones de red, asegurando siempre que el despliegue de la aplicación sea consistente y escalable.

3.1.5. Base de datos

NoSQL

Para la persistencia de datos, según lo expuesto por F. Gessert, W. Wingerath y S. Friedrich en [12], en este trabajo se ha optado por un enfoque **NoSQL** [13], más adecuado que las bases de datos relacionales tradicionales en nuestro caso de uso. Las bases de datos NoSQL se caracterizan por su flexibilidad en el modelado de datos, por su capacidad de escalado horizontal y por su buen rendimiento en escenarios con grandes volúmenes de información o estructuras no definidas de manera rígida. Este paradigma permite almacenar información de manera eficiente sin necesidad de un esquema fijo, lo que facilita la evolución de la aplicación y la integración con otros servicios.

3.1.6. Pruebas Unitarias

Con el objetivo de garantizar la calidad del software y verificar el correcto funcionamiento de cada componente de manera aislada, se han incorporado distintas herramientas para realizar las **pruebas unitarias**. Estas permiten detectar errores en etapas tempranas del desarrollo y asegurar la mantenibilidad del código.

Tox

Se ha utilizado **Tox** [14], como herramienta de automatización para la ejecución de pruebas. Tox permite gestionar la creación de entornos virtuales, instalar dependencias y lanzar pruebas de forma homogénea y sencilla.

Pytest

La herramienta principal que se ha utilizado para escribir y ejecutar pruebas es **Pytest** [15], uno de los frameworks más populares en el ecosistema Python. Pytest se caracteriza por su sencillez y simplicidad en la escritura de pruebas, por su capacidad para generar reportes detallados y por su soporte para fixtures. Esto facilita significativamente la configuración de entornos de prueba.

Además, se integra fácilmente con otras herramientas como Tox o con otros sistemas de integración continua, lo que nos ha permitido automatizar el proceso de validación del software.

Unittest

De manera complementaria, se ha usado **Unittest**, el framework de pruebas incluido en la librería estándar de Python. Aunque sus capacidades son más limitadas que las de Pytest, nos ha permitido garantizar que el código cumpla con los estándares básicos de verificación.

3.1.7. Contenedores y Despliegue

En el ámbito del despliegue, se han empleado tecnologías de **contenedores**, las cuales permiten empaquetar la aplicación junto con todas sus dependencias, garantizando así que se ejecute de forma correcta. Esto simplifica todo el proceso de desarrollo, pruebas y puesta en producción.

Docker

La herramienta principal utilizada ha sido **Docker** [16], la cual nos ha permitido definir contenedores ligeros y portables para los diferentes servicios de la aplicación. Con Docker, se ha podido aislar la ejecución del backend, los modelos de inteligencia artificial y otros componentes auxiliares, asegurando a la vez que todos ellos funcionen de manera consistente en cualquier entorno.

Dockerfile

Para construir las imágenes de los contenedores, se ha definido un **Dockerfile** [17], en el que se han especificado las dependencias, configuraciones y todos los pasos necesarios para generar la imagen final. De este modo, hemos dispuesto de un proceso reproducible y automatizable para crear los entornos de ejecución de la aplicación.

Makefile

Con el fin de simplificar y estandarizar las tareas repetitivas relacionadas con el despliegue del contenedor en AWS, se ha creado un **Makefile** [18] por cada API. Este fichero permite definir comandos abreviados para operaciones frecuentes, como la construcción de imágenes o la ejecución de contenedores. Gracias a ello, se ha conseguido agilizar el flujo de trabajo y reducir la posibilidad de errores manuales en las fases de desarrollo y despliegue.

3.2. Herramientas

Además de las tecnologías presentadas en la sección anterior, se ha empleado un conjunto de herramientas que nos han permitido desarrollar, organizar y desplegar la aplicación de forma eficiente. Estas herramientas abarcan desde la gestión del código, hasta la automatización de los procesos de despliegue.

3.2.1. Amazon Web Services

Para el despliegue de la aplicación, se ha empleado **Amazon Web Services (AWS)** [19], una de las plataformas de computación en la nube líder a nivel mundial. AWS nos ha permitido contar con un ecosistema completo de servicios que cubre todas las necesidades de la aplicación: almacenamiento, seguridad, entrega de contenido, autenticación, monitorización y orquestación de contenedores.

Gracias a ello, se ha podido garantizar **escalabilidad, alta disponibilidad y seguridad**, y reducir al mismo tiempo la complejidad de la gestión de la infraestructura.

En la sección 4.2, se describe en detalle la arquitectura de la aplicación desplegada en AWS y cómo se integran los distintos servicios. A continuación, se presentan y describen los componentes que se han utilizado:

Amazon CloudFront

Amazon CloudFront [20] es la red de distribución de contenidos (CDN) de AWS. En este proyecto, se ha utilizado para distribuir el frontend de la aplicación de manera rápida y segura, aprovechando su red global de servidores (edge locations). Esto permite que los usuarios accedan al contenido con baja latencia desde cualquier ubicación geográfica. Además, se integra de forma nativa con Amazon S3 (3.2.1.3) y con el Amazon Web Application Firewall (3.2.1.4), para añadir una capa de seguridad adicional.

Amazon API Gateway

Amazon API Gateway [21] actúa como puerta de entrada para las solicitudes dirigidas al backend. Este servicio nos ha permitido definir, exponer y gestionar las APIs de forma controlada, incluyendo mecanismos de autenticación, limitación de tráfico (throttling) y logging de peticiones. En combinación con Amazon Cognito (3.2.1.5) y el Amazon Web Application Firewall (3.2.1.4), Amazon API Gateway proporciona un flujo seguro y escalable de comunicación entre clientes y microservicios.

Amazon S3

Amazon Simple Storage Service (S3) [22] es el servicio de almacenamiento de objetos en la nube de AWS. Se ha empleado para almacenar tanto los archivos estáticos del frontend, como las imágenes subidas por el usuario y las imágenes procesadas por los modelos de inteligencia artificial. Amazon S3 proporciona durabilidad, alta disponibilidad y control detallado de accesos al bucket para los distintos servicios de AWS.

Amazon Web Application Firewall

El **Amazon Web Application Firewall (WAF)** [23] nos ha permitido proteger la aplicación frente a ataques comunes en aplicaciones web, como inyec-

ciones SQL o ataques de denegación de servicio (DDoS). En este caso, el Amazon WAF está integrado tanto en Amazon CloudFront (3.2.1.1) como en el Amazon API Gateway (3.2.1.2), para filtrar y bloquear el tráfico malicioso y los intentos de estafa, antes de que pudiera llegar a los recursos internos de la infraestructura.

Amazon Cognito

Amazon Cognito [24] es el servicio de autenticación, autorización y gestión de usuarios de AWS. Se ha utilizado para proteger el acceso de los usuarios a las APIs [25] y para gestionar de forma segura las credenciales, generando tokens JWT que posibilitan autenticar a los clientes y usuarios de la aplicación.

Amazon Lambda

Amazon Lambda [26] nos ha permitido ejecutar scripts serverless, en situaciones donde era necesario un procesamiento puntual, como la eliminación automática de imágenes de Amazon S3 (3.2.1.3). Al ser un servicio sin servidor, nos evita la obligación de tener que dejar desplegada la infraestructura y asegura un escalado automático bajo demanda, lo que reduce costes en procesos event-driven.

Amazon EventBridge

Amazon EventBridge [27] es el servicio de gestión de eventos de AWS que permite construir arquitecturas desacopladas. En esta aplicación, se ha utilizado para definir reglas que actúan como ejecutores o "triggers" de la función AWS Lambda (3.2.1.6). De esta forma, la gestión del ciclo de vida de las imágenes se realiza de manera automática, sin necesidad de procesos manuales ni programaciones externas.

Amazon Network Load Balancer

El **Amazon Network Load Balancer (NLB)** [28] se encarga de distribuir el tráfico de red entrante hacia las distintas tareas de Amazon ECS (3.2.1.10), garantizando un equilibrio de carga y tolerancia a fallos. El uso de un balanceador de carga permite manejar grandes volúmenes de tráfico con baja latencia, lo que asegura la disponibilidad de la aplicación incluso en escenarios de alta concurrencia.

Amazon Elastic Container Registry

Amazon Elastic Container Registry (ECR) [29] es el servicio donde almacenamos las imágenes Docker de la aplicación. Amazon ECR se integra de forma nativa con Amazon ECS (3.2.1.10) y nos facilita la gestión de versiones de imágenes.

Amazon Elastic Container Service

Amazon Elastic Container Service (ECS) [30] es el servicio que se ha utilizado para la orquestación de contenedores. En este caso, configurado en modalidad **Fargate**, nos permite ejecutar los microservicios sin necesidad de gestionar servidores, delegando en AWS el aprovisionamiento y escalado de la infraestructura. Gracias a Amazon ECS, se ha conseguido un entorno de ejecución altamente escalable y gestionado.

Amazon CloudWatch

Amazon CloudWatch [31] nos ha posibilitado monitorizar el estado de la aplicación y de la infraestructura. Se ha utilizado este servicio para recopilar métricas (uso de CPU, memoria, latencia de peticiones, etc.) y para centralizar logs de los contenedores. Esto nos ha facilitado el diagnóstico y la mejora del rendimiento del sistema.

Amazon DynamoDB

Amazon DynamoDB [32] [33] es un servicio de base de datos NoSQL que ofrece un modelo de almacenamiento basado en tablas, elementos y atributos. Destaca por su escalabilidad automática, su alta disponibilidad y por su integración con otros servicios de AWS. En este proyecto, se ha utilizado para almacenar la información asociada tanto a la API de orquestación como a las APIs de los distintos modelos de inteligencia artificial. Con ello se garantiza un acceso rápido, seguro y eficiente a los datos necesarios para la gestión de los microservicios desplegados.

Amazon IAM Role

Amazon Identity and Access Management (IAM) [34] nos ha permitido gestionar roles y permisos de forma granular. A través de **Amazon IAM Roles**, se ha definido qué servicios pueden interactuar entre sí (por ejemplo, dar

permisos a Amazon ECS (3.2.1.10) para leer imágenes de Amazon ECR (3.2.1.9) o a Amazon Lambda (3.2.1.6) para acceder a Amazon S3 (3.2.1.3)). Este control detallado ha sido fundamental para aplicar el principio de mínimo privilegio y mejorar la seguridad de la infraestructura.

Amazon Virtual Private Cloud

Amazon Virtual Private Cloud (VPC) [35] es un servicio de AWS que permite crear redes virtuales aisladas de manera segura dentro de la nube, ofreciendo un control completo sobre la configuración de subredes y reglas de seguridad. En este proyecto, se ha empleado Amazon VPC para organizar y proteger los distintos recursos desplegados, y se han definido subredes privadas que aseguran tanto la accesibilidad de los servicios, como la seguridad y aislamiento de los componentes internos de la arquitectura.

Amazon Security Groups

Los **Amazon Security Groups** [36] son los cortafuegos virtuales de AWS. En este despliegue, se han utilizado para definir reglas de entrada y salida en las instancias y tareas de Amazon ECS (3.2.1.10), y del Amazon NLB (3.2.1.8), restringiendo el acceso exclusivamente al tráfico necesario. De esta forma, se asegura la comunicación entre servicios internos y se protegen los endpoints públicos frente a accesos no autorizados.

Amazon Elastic Network Interfaces

Las **Amazon Elastic Network Interfaces (ENIs)** [37] son interfaces de red virtuales asociadas a los recursos dentro de la Amazon VPC. En este caso, se utilizan principalmente en Amazon ECS (3.2.1.10), pues permiten la conectividad entre los distintos componentes de la arquitectura de forma controlada y segura.

3.2.2. IDE

Visual Studio Code

Durante el desarrollo, se ha trabajado con **Visual Studio Code** [38], un editor de código ligero y altamente configurable. Gracias a su ecosistema de extensiones, se han podido integrar funcionalidades adicionales, como control de versiones con Git, depuración, ejecución de pruebas unitarias y soporte específico

para Python, Terraform y Vue.js. El uso de este IDE ha mejorado significativamente la productividad y ha facilitado la escritura de un código más limpio y mantenible.

Postman

Para la validación y prueba de los distintos endpoints desarrollados en la aplicación, se ha utilizado **Postman** [39], una de las herramientas más extendidas para el trabajo con APIs. Postman nos ha permitido simular de manera sencilla las peticiones que un cliente real realizaría y, por tanto, comprobar el correcto funcionamiento de los microservicios desplegados en AWS.

3.2.3. Control de Versiones

Git

Para el control de versiones del código fuente, se ha utilizado **Git** [40], la herramienta estándar en la industria del software. Git nos ha permitido mantener un historial completo de los cambios, trabajar en ramas y coordinar el desarrollo de diferentes componentes sin riesgo de sobrescribir trabajo previo, siguiendo una metodología Gitflow.

Además, nos ha permitido adoptar un flujo de trabajo basado en **pull requests**. Esto facilita la revisión y validación del código antes de integrarlo en la rama principal.

GitHub

Como plataforma de alojamiento del repositorio, se ha empleado **GitHub** [41], el cual además de almacenar nuestro código nos ha proporcionado un entorno para la gestión del proyecto. GitHub nos ha permitido centralizar el código y gestionar las incidencias.

3.3. Metodologías

Para el desarrollo del proyecto se ha seguido un enfoque **iterativo e incremental**, que ha permitido construir la aplicación de forma progresiva y controlada. En cada iteración se han implementado distintas funcionalidades, garantizando una evolución continua del sistema y una mejora constante en cada fase del desarrollo.

La figura 3.1 muestra el **diagrama de Gantt** correspondiente a la planificación temporal del proyecto. En él se representan las diferentes fases abordadas: desde la definición inicial y el diseño arquitectónico; hasta la implementación de la infraestructura, el backend, el frontend y los modelos de inteligencia artificial. Se muestra la secuencia temporal de cada una de las tareas.

		DIAGRAMA DE GANTT											
	TAREA	30/06/2025 - 06/07/2025	07/07/2025 - 13/07/2025	14/07/2025 - 20/07/2025	21/07/2025 - 27/07/2025	28/07/2025 - 03/08/2025	04/08/2025 - 10/08/2025	11/08/2025 - 17/08/2025	18/08/2025 - 24/08/2025	25/08/2025 - 31/08/2025	01/09/2025 - 07/09/2025	08/09/2025 - 14/09/2025	15/09/2025 - 21/09/2025
1	Planteamiento del proyecto												
2	Investigación de los recursos de AWS												
3	Diseño de la arquitectura de la aplicación												
4	Implementación de la infraestructura con IaC												
5	Desarrollo y despliegue de la API de orquestación												
6	Implementación y despliegue del frontend de la app												
7	Integración del modelo de *Upscaling*												
8	Integración del modelo de *Denoising*												
9	Integración del modelo de *Inpainting*												
10	Realización de baterías de pruebas												
11	Conclusiones obtenidas del proyecto												

Figura 3.1: Diagrama de Gantt que muestra la planificación temporal y la evolución iterativa del desarrollo del proyecto. A la izquierda, las tareas planteadas para completar el proyecto. A la derecha, la distribución del tiempo invertido en cada una de ellas.

Este diagrama refleja la evolución iterativa del trabajo, mostrando cómo los distintos componentes se han ido integrando y perfeccionando de manera progresiva hasta alcanzar la versión final del proyecto.

3.3.1. Gitflow

Para la gestión de ramas en el proyecto, se ha adoptado la estrategia **Gitflow** [42]. Se trata de un modelo ampliamente utilizado en entornos colaborativos, que define una organización clara de las ramas y facilita la integración continua de nuevas funcionalidades.

Gitflow se basa en la existencia de dos ramas principales y persistentes:

- **main**: contiene el código en estado estable y listo para producción.

- **develop**: concentra el trabajo en el desarrollo e integra las nuevas funcionalidades antes de ser desplegadas en producción.

A partir de estas ramas principales, Gitflow define diferentes ramas auxiliares con objetivos específicos:

- **feature branches**: se crean a partir de `develop` y se emplean para implementar nuevas funcionalidades de manera aislada. Una vez finalizadas, se integran de nuevo en la rama `develop`.
- **release branches**: se crean cuando el código en la rama `develop` está listo para preparar una nueva versión. Permiten realizar pruebas finales, correcciones menores y ajustes de documentación, antes de fusionar los cambios en las ramas `main` y `develop`.
- **hotfix branches**: se derivan directamente de la rama `main` y se utilizan para corregir errores críticos en producción. Una vez solucionados, los cambios se integran tanto en la rama `main` como en la rama `develop` para mantener la coherencia del repositorio.

El flujo de trabajo de Gitflow, representado en la figura 3.2, ilustra cómo estas ramas interactúan entre sí, permitiendo un desarrollo organizado, con un ciclo claro desde la implementación de funcionalidades hasta su despliegue en producción. Gracias a este enfoque, se ha podido mantener una separación efectiva entre el código estable y el código en desarrollo. Esto reduce el riesgo de introducir errores en la rama principal.

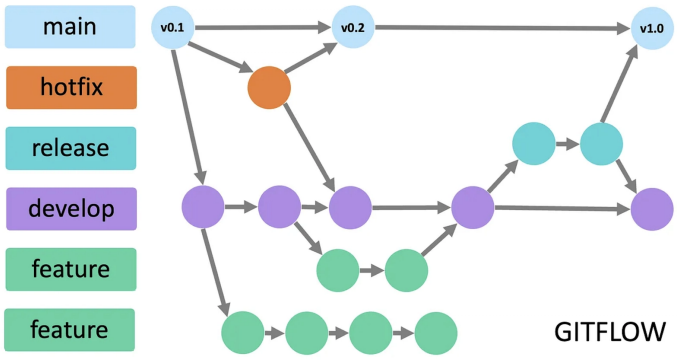


Figura 3.2: Representación del flujo de trabajo **Gitflow**. Se muestran las ramas principales (`main` y `develop`), junto con las ramas auxiliares (`feature`, `release` y `hotfix`). Cada una cumple un propósito específico: las ramas `feature` permiten desarrollar nuevas funcionalidades, las ramas `release` preparan versiones estables y las ramas `hotfix` corrigen errores críticos en producción. El diagrama refleja cómo estas ramas se integran de manera ordenada hasta llegar a versiones estables en la rama `main`. Fuente de la imagen: Truongpx. Disponible en: `https://gitflow`.

4

Descripción Informática

Este capítulo incluye la descripción del proyecto desarrollado, basado y fundamentado en las tecnologías, herramientas y metodologías expuestas en el capítulo 3. Su propósito es ofrecer una visión estructurada, clara y detallada del trabajo realizado, con el fin de proporcionar una comprensión integral del proceso seguido.

En primer lugar, en la sección 4.1 se aborda el análisis de los principales requisitos del sistema, tanto funcionales como no funcionales, que constituyen los cimientos sobre los que se ha construido la aplicación.

A continuación, en la sección 4.2, se describe la arquitectura planteada para el proyecto e incluye desde una visión general de la organización del sistema, hasta un análisis detallado de la interacción entre los distintos recursos desplegados en AWS.

Posteriormente, en la sección 4.3, se desarrolla y explica el proceso de implementación de la aplicación, describiendo los principales retos que se han afrontado y las soluciones aplicadas para garantizar la eficiencia y robustez del sistema.

En la sección 4.4, se incluyen las pruebas realizadas para verificar el correcto funcionamiento de la aplicación. Estas abarcan tanto pruebas unitarias sobre los microservicios desplegados, como pruebas de integración orientadas a validar el sistema en su conjunto, para garantizar así su fiabilidad en escenarios de uso real.

Finalmente, en la sección 4.5, se explica el procedimiento seguido para el despliegue la aplicación. Aquí se detalla cómo se han empleado contenedores Docker en combinación con diversos servicios de AWS.

4.1. Requisitos

En esta sección, se detallan los requisitos de la aplicación desarrollada. Estos requisitos incluyen tanto los aspectos funcionales, es decir, todas las funcionalidades que la aplicación debe ofrecer al usuario final, como los no funcionales, los que definen las restricciones, características de calidad y consideraciones técnicas necesarias para garantizar el correcto funcionamiento del sistema.

4.1.1. Requisitos Funcionales

- **RF1.** Como usuario, debo poder autenticarme en la aplicación a través de un sistema seguro proporcionado por Amazon Cognito.
- **RF2.** Como usuario autenticado, debo poder subir imágenes desde mi dispositivo a la aplicación, para ser procesadas.
- **RF3.** Como usuario, debo poder seleccionar desde un menú desplegable el modelo de inteligencia artificial que quiero utilizar (por ejemplo, upscaling, deconvolution, denoising, super-resolution...)
- **RF4.** Como usuario, debo poder disponer de una breve y clara explicación del modelo seleccionado, para comprender cómo el modelo va a afectar a mi imagen.
- **RF5.** Como usuario, debo poder enviar una petición, para que la API procese la imagen mediante el modelo seleccionado.
- **RF6.** Como usuario, debo poder visualizar una comparativa de la imagen original y la imagen procesada en la propia interfaz.
- **RF7.** Como usuario, debo poder descargar la imagen resultante en mi dispositivo.

4.1.2. Requisitos No Funcionales

- **RNF1.** El sistema debe permitir el despliegue y la orquestación de distintos microservicios de IA de manera modular e independiente.
- **RNF2.** El sistema debe estar diseñado para tolerar fallos, de forma que la caída de un microservicio de inteligencia artificial no comprometa el funcionamiento global de la aplicación.
- **RNF3.** La arquitectura debe ser escalable, para permitir la gestión concurrente de múltiples usuarios y peticiones simultáneas.

- **RNF4.** El despliegue de la infraestructura debe realizarse de forma automatizada mediante Terraform, garantizando siempre que sea reproducible y de fácil mantenimiento.
- **RNF5.** La aplicación debe tener una interfaz web intuitiva, amigable y accesible desde distintos dispositivos y navegadores.
- **RNF6.** Los tiempos de respuesta de la API deben ser adecuados, para permitir una interacción fluida con el usuario.
- **RNF7.** El sistema debe garantizar una comunicación segura entre el frontend y la API, mediante HTTPS.
- **RNF8.** La seguridad debe estar garantizada gracias a la autenticación con tokens JWT, empleando Amazon Cognito, y con protección frente a ataques comunes, mediante Amazon WAF.
- **RNF9.** El almacenamiento de datos temporales, como las imágenes procesadas o las subidas por el usuario, debe realizarse en S3 de manera segura.
- **RNF10.** El tiempo de vida de las imágenes almacenadas en Amazon S3 no debe ser superior a un mes. Estas deben ser eliminadas mediante una función Amazon Lambda, que utilice un trigger creado con Amazon Eventbridge.
- **RNF11.** El sistema debe registrar métricas básicas de uso y permitir la monitorización de los microservicios, mediante Amazon CloudWatch.
- **RNF12.** Los microservicios deben estar en contenedores de Docker, para asegurar la portabilidad e independencia entre componentes.
- **RNF13.** Se deben realizar pruebas unitarias y de integración, que aseguren la calidad y robustez del sistema.
- **RNF14.** La infraestructura debe estar preparada para permitir la integración de nuevos modelos de inteligencia artificial con un esfuerzo mínimo de configuración.

4.2. Arquitectura y Análisis

A continuación, se analiza la arquitectura de la aplicación desplegada en AWS, con los distintos servicios que se han integrado y utilizado para su correcto funcionamiento.

En primer lugar, se presenta un diagrama simple, que ofrece una visión general de la aplicación. Tras esto, se incluyen diagramas más específicos de la app y de las APIs, para mostrar y poder entender en detalle cómo ha sido desplegado el sistema.

4.2.1. Diagrama general del sistema

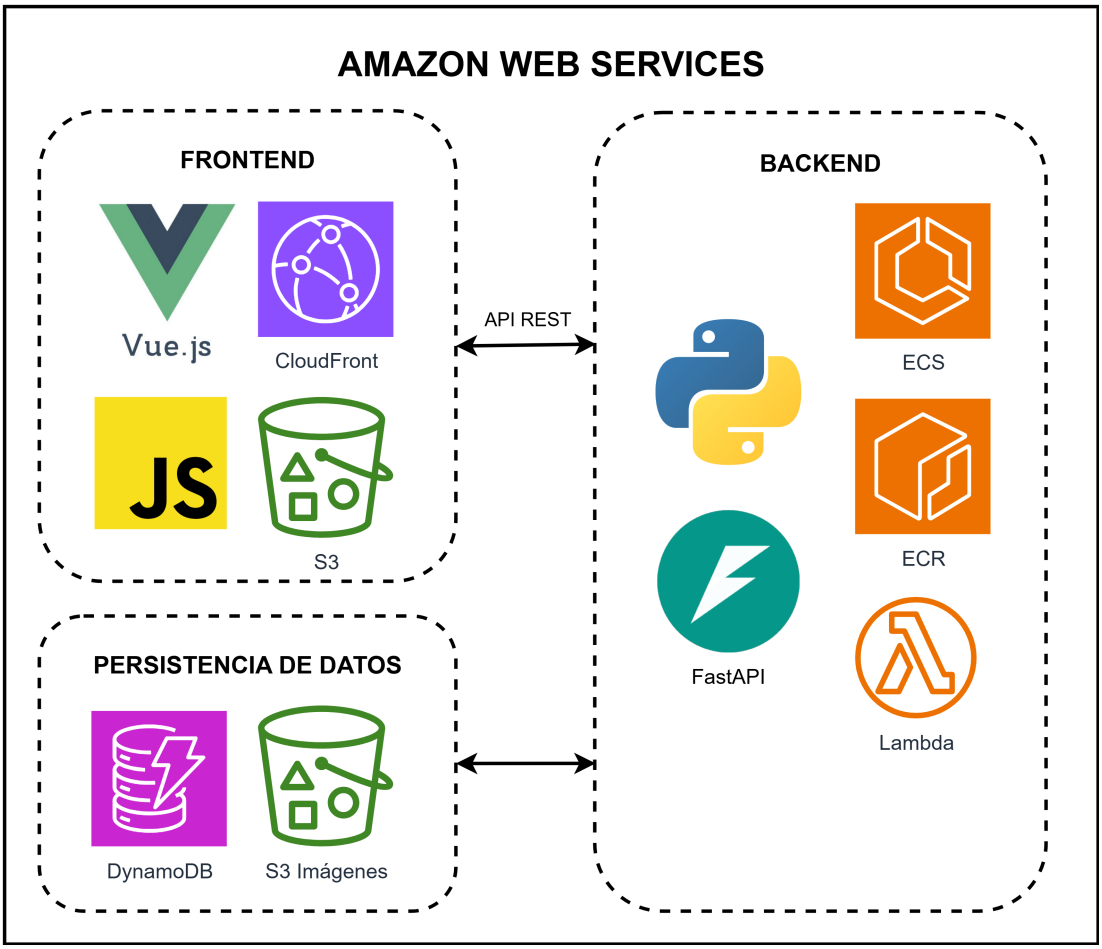


Figura 4.1: Diagrama de la arquitectura general del sistema.

La figura 4.1 muestra de forma genérica los principales recursos involucrados en la aplicación propuesta. Por un lado, el frontend, creado mediante el framework de JavaScript Vue.js, que estará almacenado de manera estática en un Amazon S3 bucket, al cual se accederá mediante Amazon CloudFront para reducir los tiempos de latencia. Este frontend será el encargado de llamar, mediante Amazon Api Gateway, a la API principal, que es la API de orquestación. Tanto esta API como el resto de APIs (de los modelos de IA) estarán disponibles en Amazon ECS.

La API de orquestación de IA se encargará, no solo de comunicarse con el resto de APIs en función del modelo que haya sido seleccionado, sino que además se responsabilizará también de otras tareas, como almacenar las imágenes resultantes en un bucket de Amazon S3 o guardar información relevante de las APIs en una base de datos no relacional (Amazon DynamoDB).

4.2.2. Diagrama de arquitectura del frontend de la aplicación

El frontend de la aplicación se ha desplegado siguiendo la arquitectura definida en la figura 4.2. Esta arquitectura basada en Amazon CloudFront, Origin Access Identity y Amazon WAF proporciona un mecanismo robusto y seguro para la distribución del frontend.

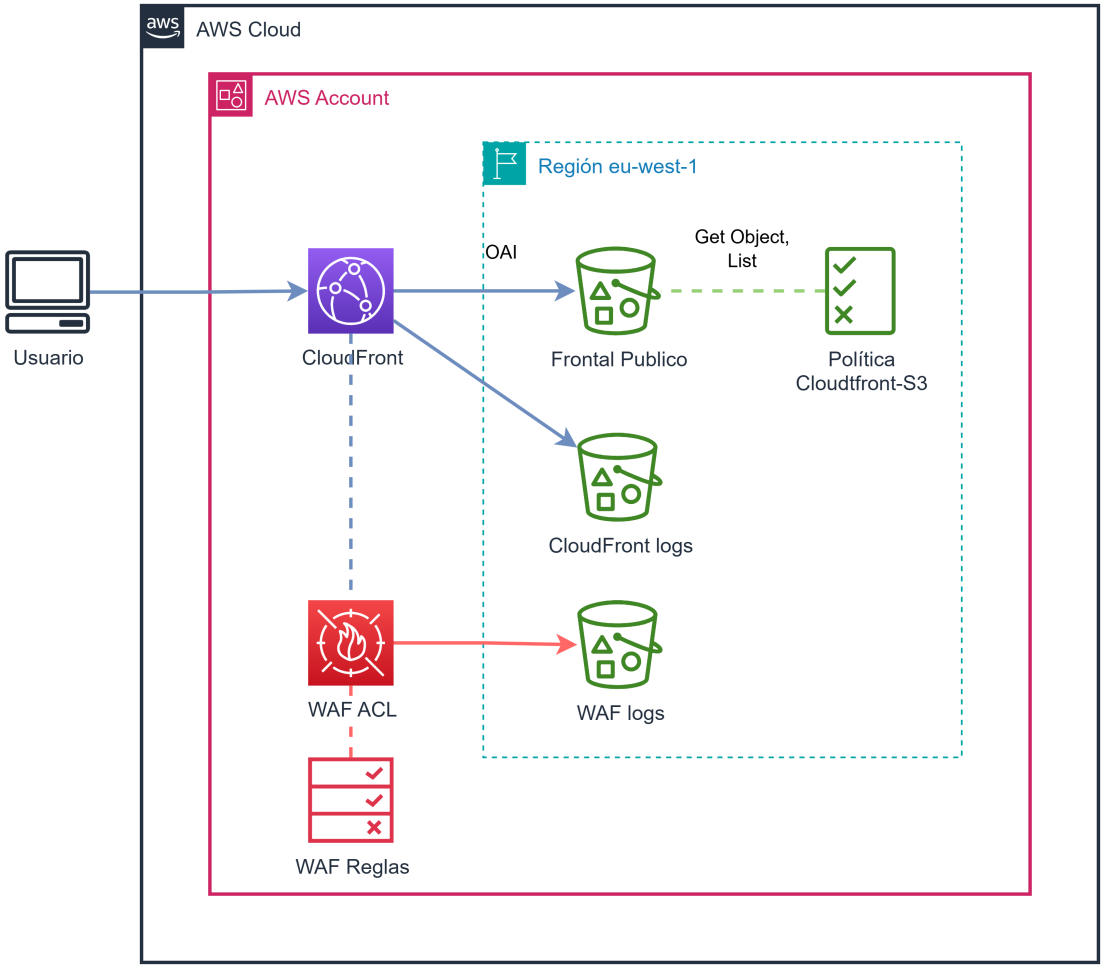


Figura 4.2: Diagrama de arquitectura del frontend de la aplicación.

Conexión entre Amazon CloudFront y Amazon S3

Tal como se observa en la figura 4.2, el flujo comienza cuando el usuario realiza una petición al frontend de la aplicación. Esta solicitud es recibida en primera instancia por **Amazon CloudFront** (3.2.1.1), que actúa como red de distribución de contenidos (CDN). Amazon CloudFront se encarga de servir los archivos estáticos del frontend de manera distribuida y con baja latencia, apoyándose en

su red global de *edge locations*. De esta manera, se mejora notablemente la experiencia de usuario, al reducirse los tiempos de respuesta, independientemente de la ubicación geográfica desde la cual se accede a la aplicación.

Una de las claves de esta arquitectura es el uso de **Origin Access Identity (OAI)**. Un OAI es una identidad virtual que se asocia a la distribución de Amazon CloudFront y permite restringir el acceso al bucket de **Amazon S3** (3.2.1.3) que contiene los archivos estáticos del frontend. En lugar de exponer el bucket de forma pública, se configura una política de acceso en Amazon S3 que solo permite a la OAI asociada a CloudFront realizar operaciones de lectura como `GetObject` o `ListBucket`. De este modo, los usuarios finales nunca acceden directamente al bucket de Amazon S3, sino que lo hacen exclusivamente a través de Amazon CloudFront. Esto refuerza significativamente la seguridad, ya que elimina accesos directos no autorizados.

Además, se ha definido un bucket correspondiente a los **Amazon CloudFront logs**, donde se almacenan las trazas de las peticiones que llegan a la distribución. Estos registros son de gran utilidad para analizar patrones de tráfico, identificar posibles errores de configuración y monitorizar el rendimiento del servicio. También resultan esenciales para realizar auditorías de seguridad, ya que permiten detectar accesos sospechosos o picos de carga anómalos.

Protección con Amazon WAF

Por otro lado, la arquitectura incorpora un componente crítico de seguridad: **Amazon WAF (Web Application Firewall)** (3.2.1.4). Este firewall de aplicaciones se integra directamente con Amazon CloudFront, aplicando un conjunto de reglas previamente configuradas, que inspeccionan el contenido de las solicitudes entrantes. Estas reglas permiten bloquear peticiones maliciosas, prevenir ataques de inyección de código y limitar intentos de denegación de servicio.

Cada decisión que toma el Amazon WAF queda registrada en un bucket dedicado a los **Amazon WAF logs**. Estos registros son especialmente relevantes para el análisis de seguridad, ya que contienen información detallada sobre las peticiones bloqueadas y sobre las razones por las que fueron rechazadas, lo que permite, por tanto, ajustar y mejorar las reglas de filtrado.

4.2.3. Diagrama de la arquitectura de la API de orquestación de los modelos de IA

La API para la orquestación de los modelos de inteligencia artificial se ha creado siguiendo la arquitectura mostrada en la figura 4.3. Esta arquitectura permite disponer de un sistema de orquestación de modelos de inteligencia artificial

robusto, seguro y escalable. La integración de servicios gestionados, junto con el uso de contenedores y funciones serverless, asegura una infraestructura flexible y con capacidad de adaptación a cambios en la carga de trabajo, manteniendo siempre la eficiencia en costes y buenas prácticas de seguridad en la nube.

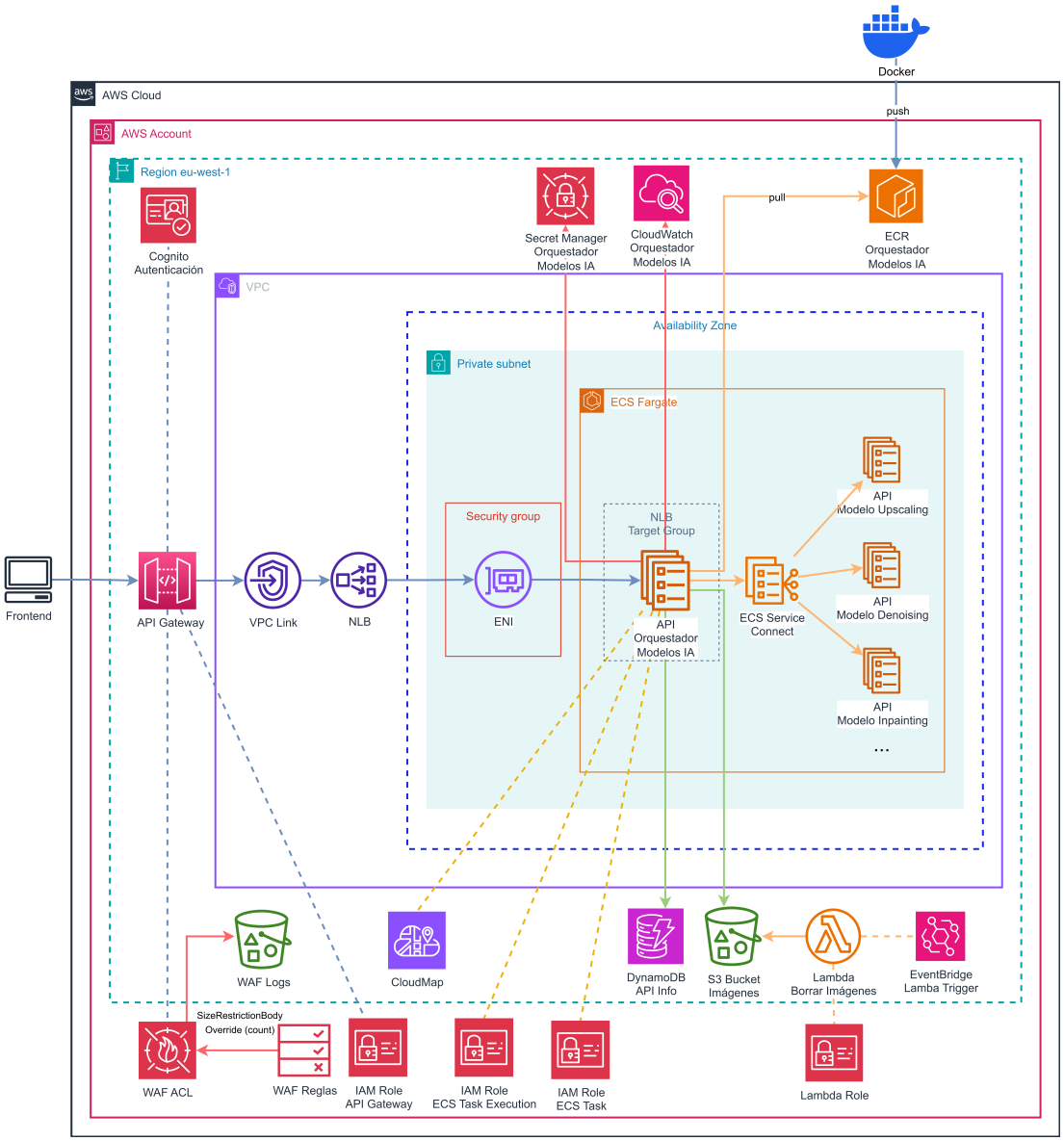


Figura 4.3: Diagrama de arquitectura de la API de orquestación de los modelos de inteligencia artificial.

Tal y como se observa en la figura 4.3, la interacción del frontend con la aplicación comienza a través de **Amazon API Gateway** (3.2.1.2), que actúa como punto de entrada a la infraestructura. API Gateway expone los diferentes endpoints de la API de orquestación de manera segura, gestiona el tráfico entrante y garantiza que las peticiones cumplan con los requisitos definidos.

Para reforzar la seguridad, Amazon API Gateway se integra con dos recursos adicionales. Por un lado, **Amazon Cognito** (3.2.1.5), que se encarga de la autenticación y autorización de los usuarios. Gracias a este servicio, únicamente los clientes debidamente autenticados pueden acceder a la API, evitando así accesos no autorizados. Por otro lado, se utiliza **Amazon WAF (Web Application Firewall)** (3.2.1.4), que permite inspeccionar y filtrar las solicitudes entrantes, y bloquea ataques comunes, como inyecciones o intentos de denegación de servicio. Para este caso, también se ha configurado una excepción concreta en las reglas del Amazon WAF, denominada *Override size restriction body*, que posibilita el envío de imágenes como parte de la petición, ya que se trata de un requisito fundamental para el posterior procesamiento en los modelos de inteligencia artificial.

Conexión entre Amazon API Gateway y Amazon NLB

Como se puede observar en la figura 4.4, una vez que la petición ha sido validada, Amazon API Gateway establece la comunicación con un **Amazon Network Load Balancer (NLB)** (3.2.1.8), mediante un **VPC Link**. Este enlace privado garantiza el tránsito seguro de las peticiones dentro de la Amazon Virtual Private Cloud (VPC) y, por tanto, evita la exposición innecesaria a Internet.

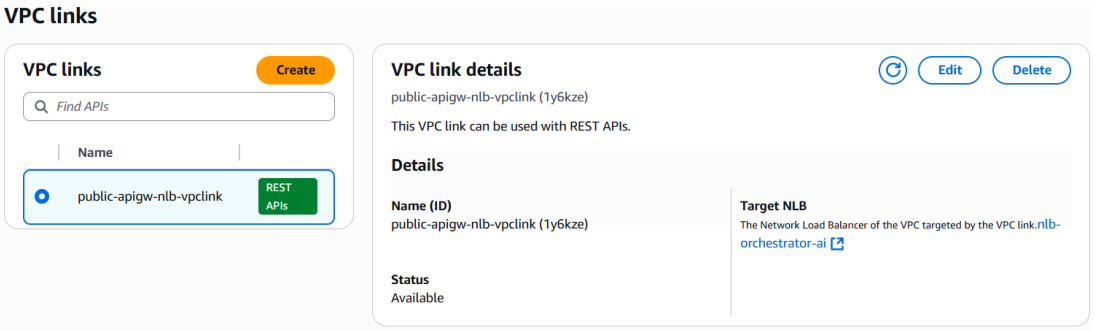


Figura 4.4: VPC Link configurado para la conexión entre el recurso de Amazon API Gateway (3.2.1.2) y el recurso Amazon Network Load Balancer (3.2.1.8) creados.

Conexión entre Amazon NLB y Amazon ECS

El Amazon Network Load Balancer (NLB) se encarga de distribuir el tráfico entrante hacia las tareas desplegadas en **Amazon ECS** (3.2.1.10), lo que garantiza así la disponibilidad y la escalabilidad de la aplicación, incluso en escenarios de alta concurrencia. Para lograrlo, tal como se puede observar en la figura 4.5, el Amazon NLB utiliza un **target group** que actúa como punto de referencia hacia los recursos de Amazon ECS, en concreto a la API encargada de la orquestación de los modelos de inteligencia artificial. Este target group no apunta directamen-

te a los contenedores, sino a la **Amazon Elastic Network Interfaces (ENI)** (3.2.1.16), que se crea de forma automática cuando Amazon ECS despliega una tarea dentro de una subred de la Amazon VPC. Estas Amazon ENI proporcionan conectividad de red a las tareas y permiten que el tráfico del Amazon NLB llegue correctamente a los servicios.

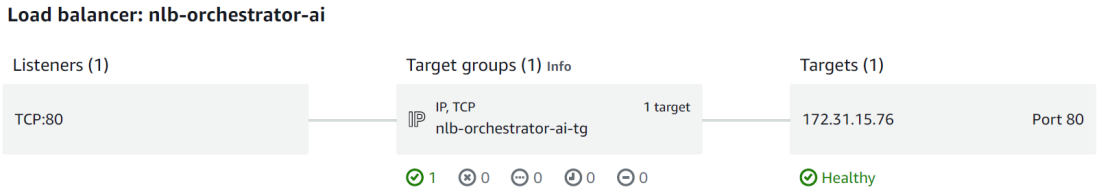


Figura 4.5: Conexión entre el Amazon NLB (3.2.1.8) y la API de orquestación. Como se puede apreciar, el target group del Amazon NLB apunta a la Amazon ENI (3.2.1.16) asociada al recurso de Amazon ECS (3.2.1.10) de la API de orquestación.

La seguridad de estas interfaces está controlada mediante **Amazon security groups** (3.2.1.15) (figura 4.6). En este caso, todas las Amazon ENI asociadas a cada tarea de Amazon ECS quedan protegidas por un Amazon security group, que define de manera explícita qué tipos de tráfico entrante y saliente están permitidos. De este modo, el Amazon NLB puede enrutar el tráfico únicamente hacia las tareas autorizadas, minimizando la exposición y reduciendo el riesgo de accesos no deseados a la API desplegada en Amazon ECS.

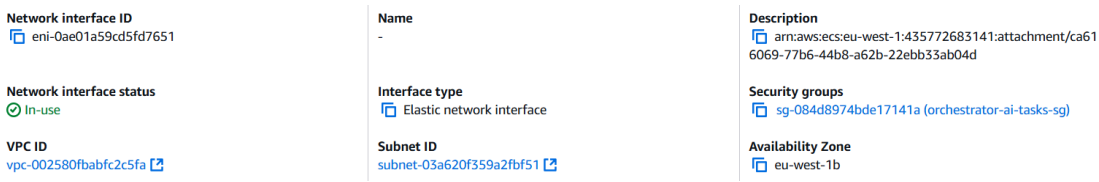


Figura 4.6: Amazon ENI (3.2.1.16) asociada a la tarea de Amazon ECS (3.2.1.10) desplegada. Como se puede observar, tiene asociado un Amazon Security Group (3.2.1.15) configurado para controlar el tráfico entrante y saliente.

Conexión dentro de Amazon ECS

Dentro del entorno de Amazon ECS, se aloja el principal servicio de la arquitectura: la **API de orquestación de modelos de inteligencia artificial**. Esta API actúa como intermediaria entre los usuarios y los distintos microservicios de modelos, gestiona las solicitudes y las redirige hacia el modelo adecuado, en función de la operación requerida. La comunicación entre microservicios se facilita mediante **Amazon ECS Service Connect**, el cual proporciona mecanismos

de enrutamiento interno, y garantiza que cada contenedor pueda comunicarse de forma eficiente con el resto.

Los microservicios desplegados en Amazon ECS corresponden a las APIs de los diferentes modelos de IA (por ejemplo, API Modelo1, API Modelo2, API Modelo3, etc.), cada uno en contenedores independientes y totalmente desacoplados, lo que permite escalar y mantener cada servicio de manera autónoma. Las imágenes Docker que contienen estos microservicios se almacenan en **Amazon Elastic Container Registry (ECR)** (3.2.1.9), desde donde son descargadas automáticamente por Amazon ECS en el momento del despliegue.

La seguridad a nivel de red también está garantizada, gracias al uso de **Amazon Elastic Network Interfaces (ENI)** y **Amazon Security Groups**. Cada tarea de Amazon ECS se ejecuta asociada a una interfaz de red elástica, que le proporciona conectividad dentro de la Amazon VPC, y se encuentra protegida por un grupo de seguridad que define las reglas de entrada y salida. De esta manera, únicamente se permite el tráfico necesario entre servicios. Además, la comunicación de ECS con el resto de recursos de la arquitectura se controla mediante los permisos definidos en **Amazon Identity and Access Management (IAM)** (3.2.1.13). Estos roles permiten, no solo que las tareas de ECS accedan de forma segura a las imágenes almacenadas en ECR, sino también que puedan acceder a los servicios auxiliares que se van a definir a continuación.

Además de los componentes principales de la arquitectura, existen otros servicios de soporte que desempeñan un papel fundamental. En primer lugar, **Amazon CloudWatch** (3.2.1.11), que se encarga de la monitorización del sistema, recopilando métricas de rendimiento y registrando logs. También se ha integrado **Amazon Secrets Manager**, que facilita el almacenamiento seguro de parámetros sensibles y evita que esos datos se incluyan en el código de la API de orquestación. En cuanto a la gestión de datos, la aplicación utiliza **Amazon DynamoDB** (3.2.1.12), como base de datos NoSQL (3.1.5.1) para almacenar información asociada a las peticiones de la API y metadatos sobre los modelos. Por último, para el manejo de archivos, se emplea **Amazon S3** (3.2.1.3), donde se almacenan las imágenes que los usuarios suben y los resultados generados por los modelos de IA.

Conexión Lambda event-driven

El ciclo de vida de las imágenes almacenadas en Amazon S3 se completa con el uso de **Amazon Lambda** (3.2.1.6), el cual se encarga de eliminar de manera automática aquellas que hayan superado un tiempo de retención definido. Amazon Lambda se ejecuta de forma *serverless*, lo que significa que no está desplegado siempre y únicamente utiliza recursos cuando es invocado. Para coordinar la ejecución de estas funciones, se utiliza **Amazon EventBridge** (3.2.1.7), que dispara

la función Lambda basándose en reglas previamente configuradas (un scheduler, por ejemplo). Esto garantiza la automatización de estas tareas de mantenimiento. La función Lambda cuenta con un rol de Amazon IAM específico, que le concede únicamente los permisos estrictamente necesarios; en este caso, la posibilidad de acceder al bucket de Amazon S3 para detectar y eliminar objetos.

4.3. Diseño e Implementación

En esta sección se detalla la implementación completa del sistema, estructurada en dos componentes principales: el **frontend**, desarrollado en `Vue.js`, y el **backend**, compuesto por microservicios implementados con `FastAPI`.

Todo el código fuente empleado para la creación íntegra de la aplicación se encuentra disponible en el repositorio de GitHub [43], accesible en el siguiente enlace: `https://github.com/ManuelGRT/aws-ai-orchestrator`.

4.3.1. Frontend con Vue.js

El desarrollo del frontend se ha realizado empleando el framework **Vue.js** (3.1.1.1), estructurando la interfaz en dos módulos principales: la pantalla de introducción deslizante y el módulo principal de análisis de imágenes. Ambos se han implementado como componentes independientes, para favorecer la modularidad.

Pantalla deslizante inicial (`IntroSlides.vue`)

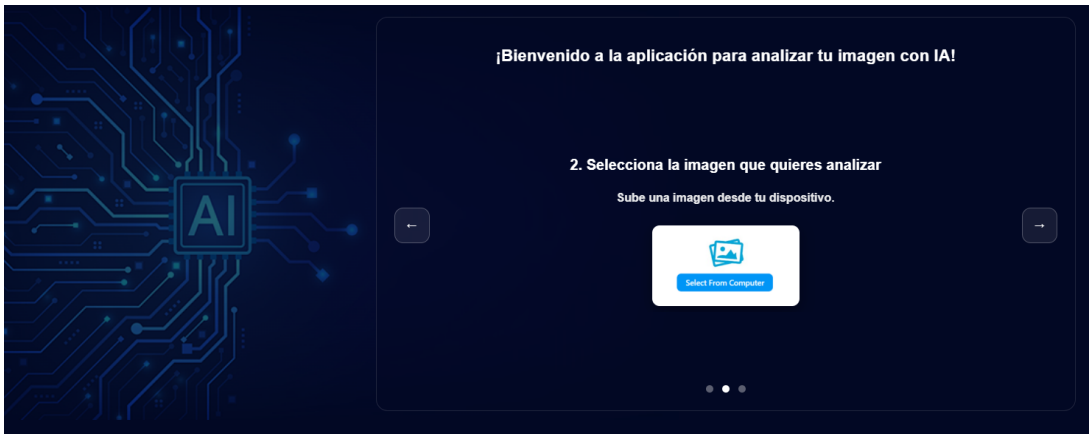
Este módulo cumple una función introductoria, guiando al usuario a través de una secuencia de pasos que explican el funcionamiento básico de la aplicación. Está diseñado mediante un sistema de diapositivas animadas que emplea la transición nativa de Vue (`<Transition name=‘‘slide’’>`), para proporcionar un desplazamiento fluido entre pantallas.

El componente define un array de `slides` con tres elementos, cada uno representando una etapa del proceso. En primer lugar, la selección del modelo (figura 4.7a), donde se explica la elección del modelo de IA a aplicar. Después, la carga de la imagen (figura 4.7b), donde se muestra cómo subir una imagen desde el dispositivo. Finalmente, en el último paso la visualización del resultado (figura 4.7c), donde se puede comparar y descargar la imagen procesada.

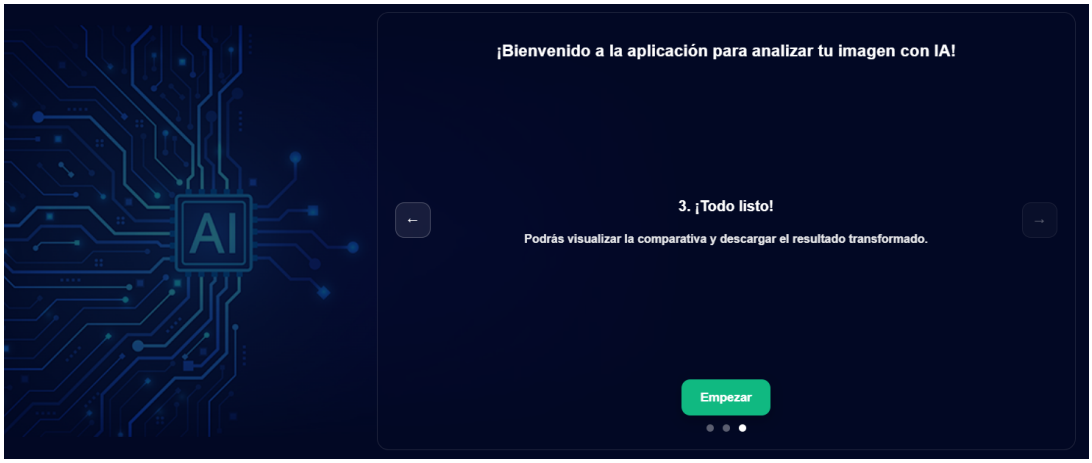
El desplazamiento entre diapositivas se gestiona mediante los métodos `next()`, `prev()` y `go(i)`, asociados a los botones laterales. Estos posibilitan tanto una navegación secuencial como el acceso directo a una diapositiva concreta. En la última

diapositiva aparece el botón `Empezar`, que emite un evento `$emit('start')` para cargar el módulo principal de análisis.

(a) Primera pantalla deslizante, en la que se explica el modelo a seleccionar.



(b) Segunda pantalla deslizante. Se muestra la selección de la imagen para analizar.



(c) Tercera pantalla deslizante, que permite empezar el proceso.

Figura 4.7: Secuencia de pantallas deslizantes introductorias de la aplicación, que guían al usuario a través del proceso inicial.

Módulo principal de análisis (`ImageAnalysis.vue`)

El módulo `ImageAnalysis.vue` constituye el núcleo funcional del frontend, ya que integra todas las interacciones principales entre el usuario y la aplicación. Este componente permite seleccionar el modelo de inteligencia artificial, subir una imagen desde el dispositivo, enviarla para su procesamiento y visualizar los resultados obtenidos de manera comparativa y dinámica.

La interfaz del módulo se organiza en diferentes bloques funcionales, comenzando con la selección del modelo de IA y la carga de la imagen. En primer lugar, el usuario puede escoger el modelo que desea aplicar a través de un desplegable (`<select>`) que incluye tres opciones: *Upscaling*, *Denoising* e *Inpainting*. Al seleccionar uno de ellos, aparece automáticamente una breve descripción del modelo elegido, ofreciendo una explicación clara sobre su función y sobre el tipo de mejora o restauración que aplicará sobre la imagen.

Una vez elegido el modelo, la aplicación permite subir un archivo local y genera de forma inmediata una vista previa en pantalla. En la figura 4.8, se muestra este primer bloque de la interfaz, donde el usuario selecciona el modelo y carga la imagen a analizar. Durante este proceso, el componente analiza la orientación de la imagen (horizontal o vertical), con el fin de adaptar dinámicamente el diseño de las tarjetas comparativas que se mostrarán al finalizar el análisis. Este comportamiento contribuye a una visualización más estética y equilibrada de los resultados.

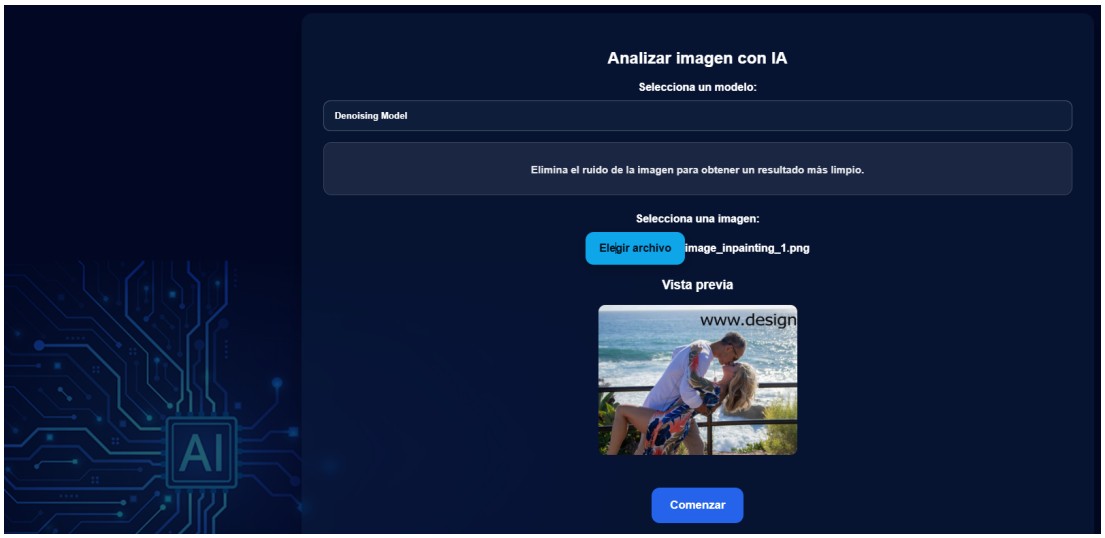


Figura 4.8: Selección del modelo y carga de la imagen. Como se puede observar, en este ejemplo se ha escogido el modelo de inpainting, encargado de eliminar el texto superpuesto. También se muestra la imagen escogida para restaurar.

Cuando el usuario pulsa el botón `Comenzar`, se inicia el flujo de comunicación con la API. En primer lugar, el frontend realiza una llamada al endpoint `/authorize`, encargado de devolver un token de sesión generado por el servicio de autenticación de **Amazon Cognito**. Este token se utiliza para autenticar todas las peticiones posteriores a la API, garantizando en consecuencia la seguridad del intercambio de datos.

Una vez obtenido el token, se envía la imagen al endpoint principal de análisis, `/image-ai-analysis`, junto con el identificador del modelo seleccionado. Tras enviar la imagen, el componente entra en un estado de espera controlado por un mecanismo de sondeo (*polling*), ejecutado cada cinco segundos durante un máximo de cinco minutos. En cada iteración, el frontend consulta el endpoint `/images/{service_name}/{image_id}`, que responde con la imagen restaurada o con un estado nulo si el proceso aún no ha finalizado. Este enfoque permite que el usuario reciba el resultado, en cuanto el modelo de IA complete su ejecución en el backend, evitando bloqueos o recargas manuales de la página. La figura 4.9 muestra el esquema de endpoints, a los que el frontend realiza peticiones durante el proceso completo.

Name	Status	Type	Initiator	Size	Time
authorize/	200	xhr	index-DNhbuX4i.js	4....	714 ms
image-ai-analysis	200	xhr	index-DNhbuX4i.js	0....	589 ms
ffcfaf72-b137-4a12-9221-2019149b7...	CORS er...	xhr	index-DNhbuX4i.js	0....	222 ms
ffcfaf72-b137-4a12-9221-2019149b7...	CORS er...	xhr	index-DNhbuX4i.js	0....	217 ms
ffcfaf72-b137-4a12-9221-2019149b7...	CORS er...	xhr	index-DNhbuX4i.js	0....	206 ms
ffcfaf72-b137-4a12-9221-2019149b7...	CORS er...	xhr	index-DNhbuX4i.js	0....	287 ms
ffcfaf72-b137-4a12-9221-2019149b7...	200	xhr	index-DNhbuX4i.js	54...	324 ms

Figura 4.9: Registro de los endpoints de la API de orquestación a los que llama el frontend. Como se puede observar, primero se llama al endpoint `/authorize` para obtener el token de autenticación de Amazon Cognito. Con este token se realizan el resto de llamadas, tanto a `/image-ai-analysis`, para enviar a analizar la imagen, como a `/images/{service_name}/{image_id}`, para obtener la imagen restaurada.

Cuando finalmente la API devuelve la imagen procesada, el componente genera un objeto URL a partir del flujo de bytes recibido y lo asigna a la variable `resultUrl`. En ese momento, la vista se actualiza de manera automática y muestra la comparativa entre la imagen original y la transformada, ambas dispuestas en tarjetas paralelas con un diseño simétrico y moderno. Además, se habilita un botón que permite descargar el resultado procesado directamente en formato `.png`, completando así el ciclo de análisis. En la figura 4.10, se observa la disposición final de la interfaz con ambas imágenes y la opción de descarga.

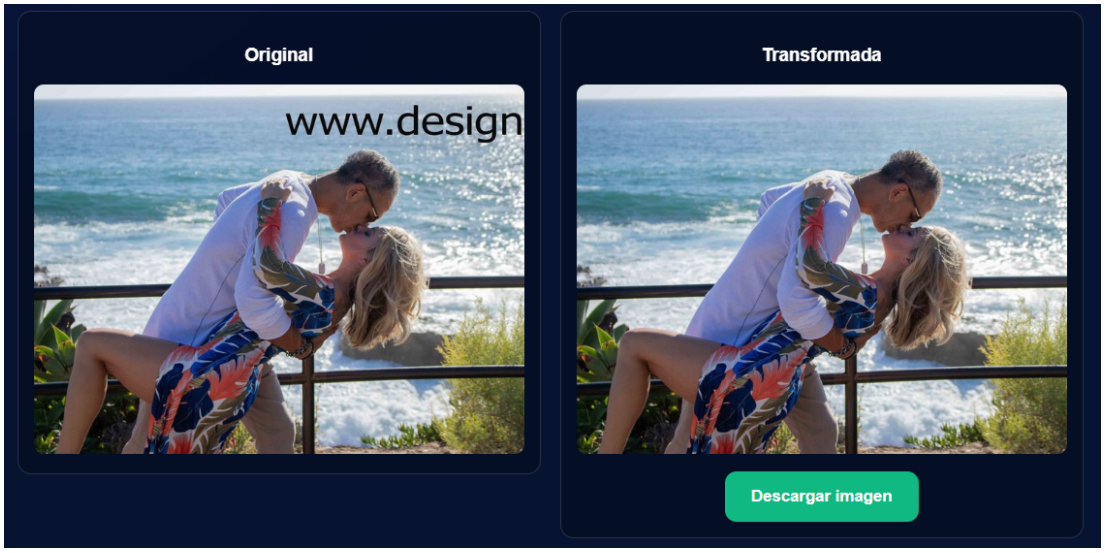


Figura 4.10: Comparativa final entre la imagen original y la imagen restaurada, empleando el modelo de inpainting. A la izquierda, se puede observar la imagen original. A la derecha, la imagen transformada, en la que desaparece el texto de la imagen.

Desde el punto de vista de la gestión del estado, el componente utiliza referencias (`ref`) para mantener sincronizados los distintos elementos de la interfaz. Variables como `selectedModel`, `selectedFile` y `originalPreviewUrl` controlan la entrada de datos del usuario, mientras que `resultUrl`, `loading` y `waitingResult` describen los estados intermedios del proceso. Asimismo, el atributo `error` permite notificar al usuario, en caso de fallos de comunicación o de tiempos de espera excedidos.

4.3.2. Microservicios con FastAPI

El backend de la aplicación se ha desarrollado siguiendo una arquitectura modular basada en microservicios, donde cada microservicio corresponde a una API independiente responsable de un componente o modelo de la aplicación. Esta estructura favorece la escalabilidad, el mantenimiento y la posibilidad de desplegar, actualizar o escalar cada servicio de forma individual, sin afectar al resto del sistema. Cada microservicio ha sido implementado con FastAPI (3.1.2.1).

Para contextualizar los modelos de inteligencia artificial utilizados, en el apéndice B se incluye una introducción general a la inteligencia artificial y sus principales subáreas, finalizando con una descripción detallada de las redes neuronales convolucionales (CNN) y de la arquitectura U-Net, ampliamente utilizada en tareas de procesamiento de imágenes.

API de orquestación de los modelos IA

La **API de orquestación** constituye el núcleo del sistema backend. Su función es actuar como intermediaria entre el frontend y los distintos servicios de IA desplegados, así como gestionar la comunicación con el resto de recursos de AWS (Amazon S3 y Amazon Cognito, entre otros).

Esta API contiene tres endpoints principales:

- `/authorize`: obtiene el token de sesión de Amazon Cognito, para la autenticación de usuario.
- `/image-ai-analysis`: recibe la imagen enviada desde el frontend y la envía al modelo de IA correspondiente, para su análisis.
- `/images/{service_name}/{image_id}`: devuelve la imagen procesada, una vez ha sido restaurada y almacenada en Amazon S3.

El endpoint `/image-ai-analysis` implementa un mecanismo de procesamiento en segundo plano mediante `background tasks`, lo que permite liberar la petición HTTPS y devolver una respuesta inmediata al usuario, mientras el análisis de la imagen continúa ejecutándose de forma asíncrona. El fragmento de código mostrado en el código 4.1 ilustra este comportamiento. Esta estructura permite mantener la API principal ligera y eficiente, sin depender del tiempo de ejecución de los modelos de IA, que pueden ser más costosos computacionalmente.

Código 4.1: Función llamada en background task, para analizar la imagen con un modelo de IA

```
async def analyze_new_image(image_bytes: bytes, image_id: str,
    service_name: str, logger: LoggingManager):
    """
    Analiza la imagen y guarda el resultado
    """
    try:
        logger.info(f"Analyzing image with AI service
            {service_name}...")
        # Llamar a la API del modelo en un thread pool
        upscale_api = ModelAiApi()
        response_image_bytes = await run_in_threadpool(
            lambda: upscale_api.analyze_image(
                image_bytes=image_bytes,
                image_id=image_id,
                service_name=service_name,
                logger=logger
            )
        )
```

```
17          if response_image_bytes is None:
18              raise HTTPException(status_code=400, detail="Error
                    Getting Api Info")
19
20          logger.info("Image analyzed successfully.")
21
22          # Guardar la imagen procesada en S3
23          await save_new_image(
24              image_bytes=response_image_bytes,
25              image_id=image_id,
26              image_prefix=service_name,
27              logger=logger,
28              content_type="image/png"
29          )
30
31      except Exception as error:
32          logger.error(f"Error during analyze image: {error}")
33          raise HTTPException(status_code=400, detail="Error
                during analyze image")
```

En lo relativo a la persistencia de datos, todos los `logs` presentes a lo largo del código se centralizan en **Amazon CloudWatch**, lo que permite monitorizar en tiempo real la actividad de las APIs, detectar posibles errores y analizar los tiempos de ejecución de cada proceso. Cada microservicio dispone de un `LoggingManager` propio encargado de enviar la información de registro a CloudWatch, donde los eventos pueden consultarse fácilmente a través de la consola de AWS.

2025-09-30T23:44:17.044+02:00	1759268657044 [INFO] timestamp: 2025-09-30T21:44:17.044230+00:00 | request_id: 318a8346-a7f2-4885-833e-b3e8922d9357 | image_id: 02948061-530d-40ee-b36b-985a5d315d13 | logging_message: Starting ai orchestrator process... [api.routers.logging] X-Request-ID=318a8346-a7f2-4885-833e-b3e8922d9357
2025-09-30T23:44:17.044+02:00	1759268657045 [INFO] timestamp: 2025-09-30T21:44:17.044457+00:00 | request_id: 318a8346-a7f2-4885-833e-b3e8922d9357 | image_id: 02948061-530d-40ee-b36b-985a5d315d13 | logging_message: Starting orchestrator process for image 02948061-530d-40ee-b36b-985a5d315d13 [api.routers.logging] X-Request-ID=318a8346-a7f2-4885-833e-b3e8922d9357
2025-09-30T23:44:17.044+02:00	1759268657045 [INFO] timestamp: 2025-09-30T21:44:17.044702+00:00 | request_id: 318a8346-a7f2-4885-833e-b3e8922d9357 | image_id: 02948061-530d-40ee-b36b-985a5d315d13 | logging_message: Saving original image... [api.routers.logging] X-Request-ID=318a8346-a7f2-4885-833e-b3e8922d9357
2025-09-30T23:44:17.044+02:00	1759268657045 [INFO] timestamp: 2025-09-30T21:44:17.044820+00:00 | request_id: 318a8346-a7f2-4885-833e-b3e8922d9357 | image_id: 02948061-530d-40ee-b36b-985a5d315d13 | logging_message: Begin image upload to S3 bucket... [api.routers.logging] X-Request-ID=318a8346-a7f2-4885-833e-b3e8922d9357
2025-09-30T23:44:17.165+02:00	1759268657165 [INFO] timestamp: 2025-09-30T21:44:17.165301+00:00 | request_id: 318a8346-a7f2-4885-833e-b3e8922d9357 | image_id: 02948061-530d-40ee-b36b-985a5d315d13 | logging_message: Ending image upload to S3 bucket... [api.routers.logging] X-Request-ID=318a8346-a7f2-4885-833e-b3e8922d9357
2025-09-30T23:44:17.167+02:00	1759268657167 [INFO] timestamp: 2025-09-30T21:44:17.166882+00:00 | request_id: 318a8346-a7f2-4885-833e-b3e8922d9357 | image_id: 02948061-530d-40ee-b36b-985a5d315d13 | logging_message: Original image saved successfully. [api.routers.logging] X-Request-ID=318a8346-a7f2-4885-833e-b3e8922d9357
2025-09-30T23:44:17.167+02:00	1759268657167 [INFO] timestamp: 2025-09-30T21:44:17.167109+00:00 | request_id: 318a8346-a7f2-4885-833e-b3e8922d9357 | image_id: 02948061-530d-40ee-b36b-985a5d315d13 | logging_message: Finishing ai orchestrator process... [api.routers.logging] X-Request-ID=318a8346-a7f2-4885-833e-b3e8922d9357
2025-09-30T23:44:17.167+02:00	1759268657167 [INFO] timestamp: 2025-09-30T21:44:17.167187+00:00 | request_id: 318a8346-a7f2-4885-833e-b3e8922d9357 | image_id: 02948061-530d-40ee-b36b-985a5d315d13 | logging_message: Ai orchestrator image process time: 0.12272882461547852 [api.routers.logging] X-Request-ID=318a8346-a7f2-4885-833e-b3e8922d9357

Figura 4.11: Visualización de los logs de ejecución de la API de orquestación, en Amazon CloudWatch.

La figura 4.11 muestra un ejemplo de los registros generados en uno de los microservicios, en el cual se pueden observar los mensajes de información que se producen durante la ejecución de las operaciones.

De forma complementaria, la información relativa a las peticiones realizadas a las distintas APIs se persiste en Amazon DynamoDB. Para no bloquear el flujo de ejecución de la API principal, el envío de esta información también se realiza mediante una tarea en segundo plano (`background task`).

El siguiente fragmento de código 4.2 muestra la implementación de esta tarea asíncrona, en la cual se construye un objeto `OrchestratorApiPersistance` con los datos relevantes y se envía a Amazon DynamoDB, mediante la función `upload_api_persistance`:

Código 4.2: Background Task para guardar la información de la llamada a un endpoint de la API de orquestación en Amazon DynamoDB, en caso de que no haya habido error.

```
dynamodb = DynamoDB()

background_tasks.add_task(
    dynamodb.upload_api_persistance,
    logger=logger,
    data_api=OrchestratorApiPersistance(
        request_id=context.get(header_keys.HeaderKeys.request_id),
        api_id="orchestrator_ai_api",
        image_id=str(image_id),
        response_latency=int(response_time*1000),
        request_datetime=datetime.utcnow().isoformat(),
        http_method=request.method,
        resource_path=str(request.url.path),
        status="200",
        error_message=None
    )
)
```

Finalmente, en lo referente a las imágenes procesadas, el sistema almacena tanto las imágenes originales subidas por el usuario como las resultantes de los modelos de IA, en Amazon S3, el servicio de almacenamiento de objetos de AWS. Se ha definido una estructura jerárquica de carpetas, que clasifica las imágenes en función de su origen y del modelo que las ha procesado (figura 4.12). Esta organización permite mantener una trazabilidad clara de los resultados obtenidos.

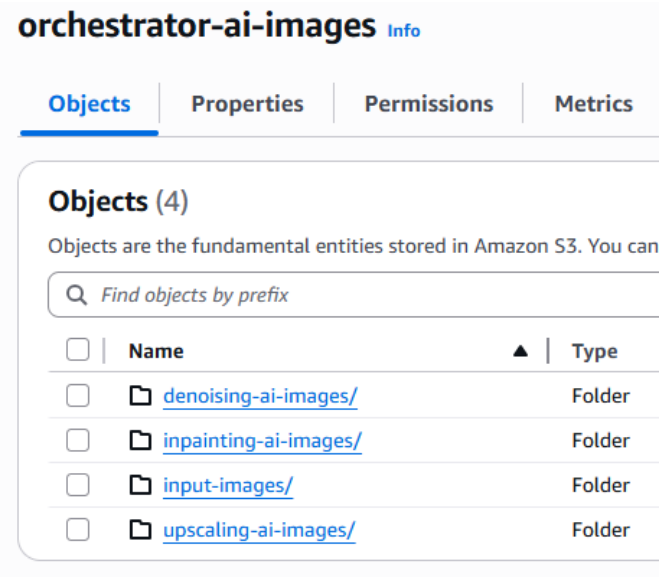


Figura 4.12: Estructura de carpetas definida para la persistencia de las imágenes enviadas y procesadas.

API de Denoising

La API de **Denoising** tiene como objetivo eliminar el ruido presente en las imágenes degradadas, obteniendo una versión restaurada de la imagen más limpia. Para ello, se emplea la librería `DeepInv` (3.1.3.2), que permite implementar modelos de restauración basados en arquitecturas profundas de redes neuronales.

El modelo seleccionado es **DRUNet** (Deep Residual U-Net), una arquitectura desarrollada por Zhang et al. en [44], ampliamente utilizada en tareas de denoising, por su capacidad para combinar convoluciones profundas con conexiones residuales dentro de una estructura tipo U-Net. Este modelo destaca por tener la capacidad de generalizar frente a distintos niveles de ruido y tipos de degradación.

El endpoint principal, `/analyze-image`, ejecuta dos fases fundamentales:

1. Estimación automática del ruido: se calcula el parámetro σ, que representa la desviación estándar del ruido presente en la imagen.
2. Reconstrucción de la imagen: la imagen ruidosa se procesa con el modelo `DRUNet`, que aprende a mapear directamente la imagen degradada hacia una versión limpia, utilizando el valor estimado de σ como guía durante la inferencia.

Una vez estimado el valor del ruido σ, la imagen se convierte a un tensor y se procesa mediante el modelo `DRUNet`, previamente cargado y configurado en modo

evaluación. Este modelo ya preentrando aplica un proceso de restauración que elimina el ruido manteniendo los bordes y texturas originales de la imagen. El siguiente fragmento de código 4.3 muestra la parte de la API donde se ejecuta el modelo y se obtiene la imagen restaurada:

Código 4.3: Aplicación del modelo DRUNet y reconstrucción de la imagen restaurada.

```
x = to_tensor(img).unsqueeze(0).to(DEVICE)
sigma_norm = sigma / 255.0

logger.info(f"Denoising image with sigma = {sigma:.2f}...")
with torch.no_grad():
    y = DENOISER(x, sigma=sigma_norm).clamp(0, 1)
logger.info(f"Denoising completed")
```

En el código 4.3, se observa cómo la imagen original se convierte a un tensor mediante `torchvision.transforms.ToTensor()`, normalizando sus valores en el rango $[0, 1]$. A continuación, se calcula el valor normalizado del ruido `sigma_norm`, que se pasa como argumento al modelo `DRUNet`.

El modelo, ejecutado en modo `no_grad()`, genera una versión restaurada de la imagen (`y`) que posteriormente se recorta al rango válido y se convierte de nuevo a formato `PIL`, para su almacenamiento temporal en memoria.

El resultado final se muestra en la figura 4.13. Se puede apreciar la reducción significativa del ruido, manteniendo los detalles estructurales de la imagen.

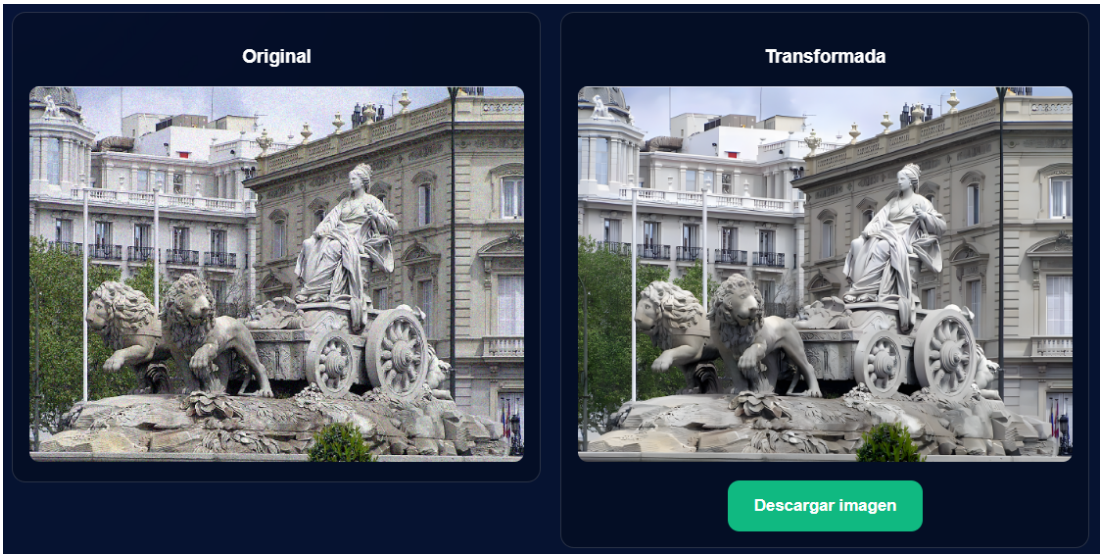


Figura 4.13: Comparativa final entre la imagen original y la imagen restaurada, empleando el modelo DRUNet. A la izquierda, se puede observar la imagen original. A la derecha, la imagen transformada, en la que se ha logrado eliminar el ruido de la imagen.

Durante el despliegue de esta API, se presentaron diversas incompatibilidades entre las versiones de `torch`, `torchvision` y `deepinv`. Para resolver esta cuestión, se fijaron versiones estables y mutuamente compatibles en el fichero `requirements.txt`, estableciendo las siguientes dependencias:

`torch==2.8.0`, `torchvision==0.23.0`, `deepinv==0.3.4`

API de Inpainting

La API de **Inpainting** tiene como finalidad reconstruir de manera coherente las regiones ausentes de una imagen, debido a que están superpuestas por texto. Por lo tanto, esta API se emplea específicamente para la eliminación automática de texto en imágenes, de modo que el sistema detecta las zonas con texto, las borra y las rellena generando contenido visualmente consistente con el entorno.

El funcionamiento de la API se basa en la combinación de dos etapas fundamentales:

1. Detección automática de texto mediante OCR (Optical Character Recognition). Se utiliza la librería `paddleOCR` (3.1.3.4), que analiza la imagen, y devuelve las coordenadas y la confianza de cada región de texto detectada. A partir de esta información, se genera una máscara binaria, donde los píxeles correspondientes al texto se marcan en blanco, mientras que el resto de la imagen permanece en negro.

2. Reconstrucción de la imagen mediante el modelo LAMA (Large Mask Inpainting Model). Una vez generada la máscara, se aplica el modelo `LAMA`, integrado a través de la librería `lama_cleaner` (3.1.3.5). Este modelo, basado en redes neuronales convolucionales profundas, es capaz de predecir los píxeles ausentes en grandes áreas de la imagen, generando una reconstrucción realista y coherente con su contexto.

El endpoint principal, `/analyze-image`, recibe la imagen desde el frontend y crea la máscara de texto con el detector OCR. Una vez generada la máscara binaria con las regiones a eliminar, la imagen se envía al modelo de inpainting `LAMA`, para que realice el proceso de restauración.

Código 4.4: Aplicación del modelo LAMA, para la reconstrucción de la imagen.

```
try:
    logger.info(f"Inpainting image...")
    img_np = np.array(pil_img)
    result = lama(img_np, mask, cfg)
    result = normalize_result(result)
    logger.info(f"Image inpainted successfully")
```

```
except Exception as e:
    logger.error(f"Inpainting error: {e}")
    raise HTTPException(status_code=500, detail=f"Inpainting
        error: {e}")
```

En el código 4.4 se observa cómo se transforma la imagen a `NumPy` y cómo se aplica el modelo `LAMA` junto con la máscara de texto y la configuración del modelo (`cfg`), donde se definen los parámetros de inferencia. Finalmente se normaliza la salida, mediante la función `normalize_result()`, para asegurar que la imagen final esté en formato RGB con valores enteros de 8 bits.

En la figura 4.14, se presenta un ejemplo del proceso de eliminación automática de texto sobre una imagen real. La primera muestra corresponde a la imagen original, mientras que la segunda representa la versión restaurada por el modelo LAMA.

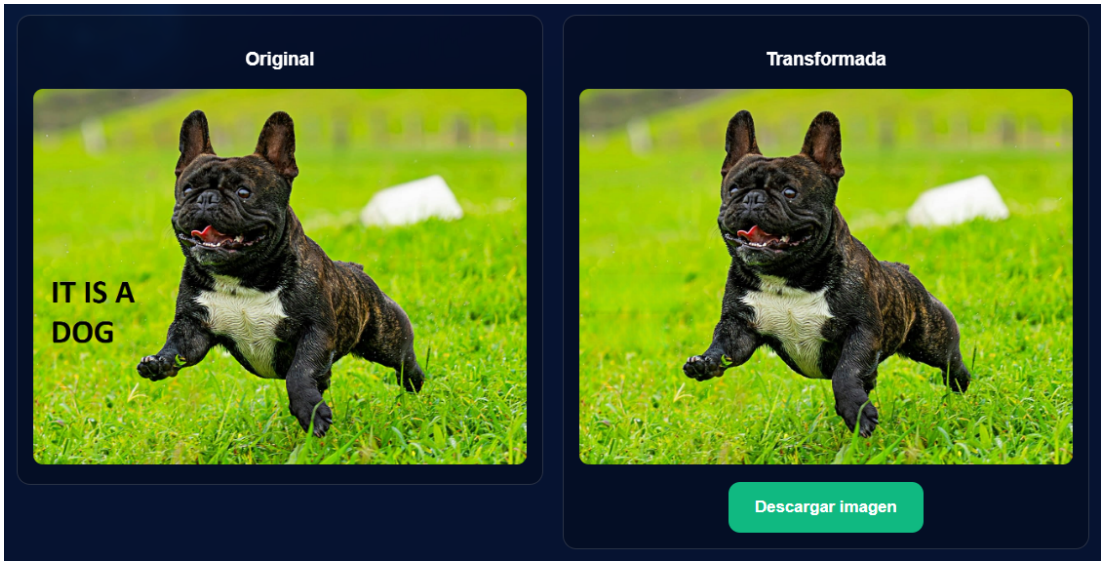


Figura 4.14: Comparativa final entre la imagen original y la imagen restaurada empleando el modelo de inpainting. A la izquierda, se puede observar la imagen original. A la derecha, la imagen transformada, en la que ha desaparecido el texto de la imagen.

Durante la fase de despliegue, esta API presentó varios retos técnicos relacionados con la compatibilidad entre librerías. El paquete `lama_cleaner` depende de `diffusers` y de `huggingface-hub` para la gestión del modelo, pero instala automáticamente versiones más recientes que no siempre mantienen compatibilidad con los modelos de inferencia antiguos. Para resolver este problema, fue necesario fijar versiones específicas y forzar su instalación sin dependencias (`--no-deps`), para evitar que las librerías se actualizaran de forma recursiva y rompieran el entorno.

El fragmento que se muestra a continuación (código 4.5) refleja la configuración final utilizada en el `Dockerfile`, donde se instalaron manualmente las versiones estables y compatibles entre sí:

Código 4.5: Instalación controlada de dependencias para la API de Inpainting.

```
RUN pip install --no-cache-dir huggingface-hub==0.12.1 \
 && pip install --no-cache-dir diffusers==0.16.1 --no-deps \
 && pip install --no-cache-dir lama-cleaner==1.2.5 --no-deps
```

API de Upscaling

La API de **Upscaling** tiene como objetivo aumentar la resolución de una imagen de entrada sin que exista una pérdida de calidad perceptible, generando una versión más nítida y detallada de la misma.

Para esta tarea, se integra el modelo **Real-ESRGAN** (Enhanced Super-Resolution Generative Adversarial Network), implementado a través de la librería `Real-ESRGAN NCNN` (3.1.3.3). Este modelo, basado en una arquitectura GAN, genera resultados realistas y preserva detalles finos, mediante un discriminador adversarial que distingue entre regiones naturales y artefactos de interpolación.

El endpoint principal `/analyze-image` recibe una imagen y la procesa directamente mediante el modelo `Real-ESRGAN`, devolviendo una versión con una resolución incrementada. El procedimiento se resume en el siguiente fragmento:

Código 4.6: Upscaling de una imagen utilizando Real-ESRGAN.

```
logger.info("Upscaling image...")
realesrgan = request.app.state.realesrgan
upscaled = realesrgan.process_pil(img)
logger.info("Image upscaled successfully")
```

En el código 4.6, se puede observar cómo se accede al modelo previamente cargado en el estado de la aplicación (`request.app.state.realesrgan`) y se ejecuta el método `process_pil()`, que procesa la imagen en formato `PIL` y devuelve una nueva imagen con una resolución superior.

El resultado de este proceso se muestra en la figura 4.15, en la que se aprecia la mejora significativa de resolución y nitidez lograda por el modelo.

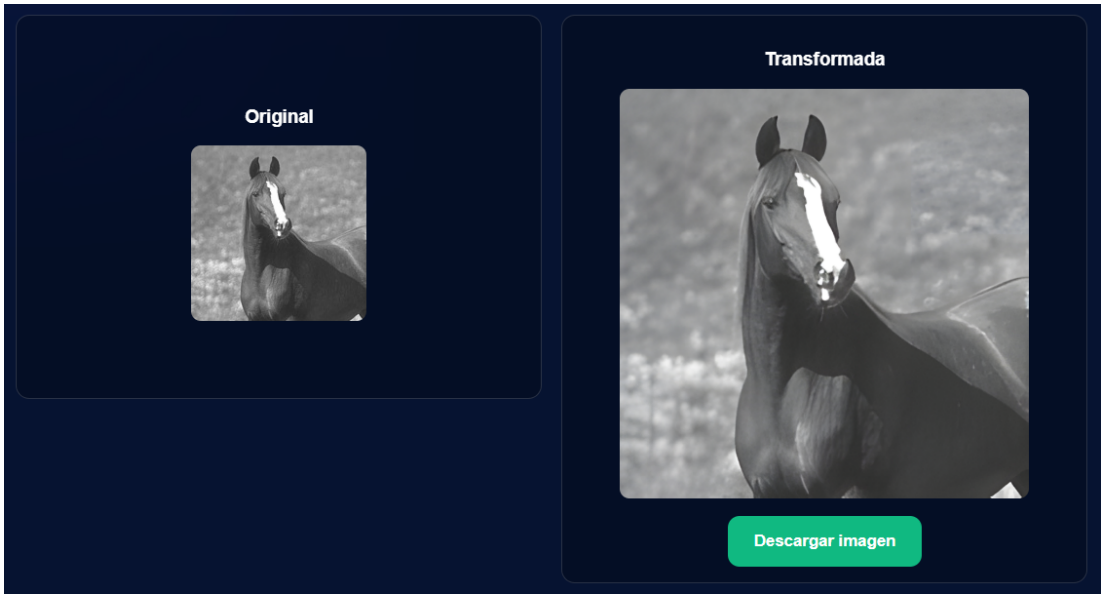


Figura 4.15: Comparativa final entre la imagen original y la imagen restaurada, empleando el modelo de Upscaling. A la izquierda, se puede observar la imagen original. A la derecha, la imagen transformada, en la cual la imagen tiene más resolución.

Durante el despliegue en producción, surgieron diversos problemas relacionados con la compatibilidad de librerías, especialmente `opencv-python-headless` y las dependencias de Vulkan requeridas por el motor NCNN. Para resolverlos, fue necesario incluir las librerías del sistema en la imagen Docker, como se muestra en el código 4.7.

Código 4.7: Instalación de dependencias del sistema y librerías para el modelo Upscaling.

```
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir
       "opencv-python-headless==4.10.0.84"

COPY api/ ./${API_NAME}/api/
COPY ${REQUIREMENTS_FILE} ./${API_NAME}/

RUN apt-get update && apt-get install -y
   --no-install-recommends \
    libvulkan1 mesa-vulkan-drivers libomp5 ca-certificates \
 && rm -rf /var/lib/apt/lists/*
```

4.4. Pruebas

En esta sección, se presentan las principales pruebas realizadas con el objetivo de garantizar que la aplicación desarrollada cumple con los requisitos de estabilidad, robustez y fiabilidad. Para una mejor organización, las pruebas se han dividido en tres categorías: pruebas de integración, pruebas de sistema y pruebas unitarias.

4.4.1. Pruebas de Integración

Las pruebas manuales de integración se han llevado a cabo para verificar la correcta comunicación entre los distintos microservicios que conforman la arquitectura. Para ello, se ha empleado la plataforma Postman (3.2.2), desde la cual se realizaron múltiples llamadas a la API de orquestación. Estas pruebas permitieron comprobar que la API principal se integra adecuadamente con las APIs de los modelos de inteligencia artificial desplegados.

Las llamadas realizadas a través de Postman han sido recogidas en el fichero `orchestrator_api.postman_collection.json` en formato JSON. Este fichero se puede consultar dentro de la carpeta `backend/`, en el repositorio de GitHub [43]. En la figura 4.16, se muestra un ejemplo de una de estas llamadas.

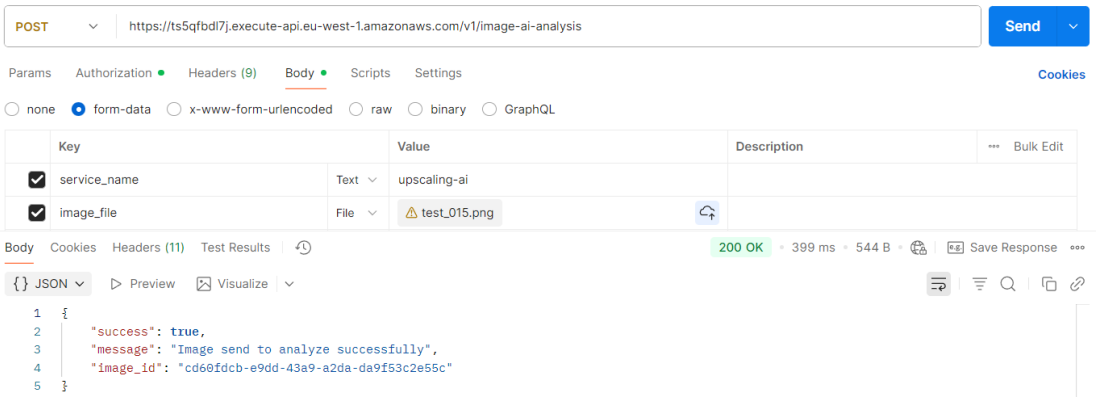


Figura 4.16: Ejemplo de llamada a la API de orquestación desplegada en AWS, utilizada para comprobar la correcta integración con el resto de microservicios desplegados en Amazon ECS.

4.4.2. Pruebas de Sistema

Las pruebas manuales de sistema se han orientado a validar el comportamiento de la aplicación en su conjunto, evaluando distintos escenarios de uso con el objetivo de asegurar la fiabilidad de la solución.

Para ejemplificar las pruebas de sistema que se realizaron, en la figura 4.17 se muestra la carpeta de Amazon S3 donde se registran las imágenes subidas para analizar durante el último mes. En total, se realizaron pruebas con 52 imágenes, lo que permitió comprobar el correcto funcionamiento y comunicación entre el frontend, el backend y la arquitectura desplegada.

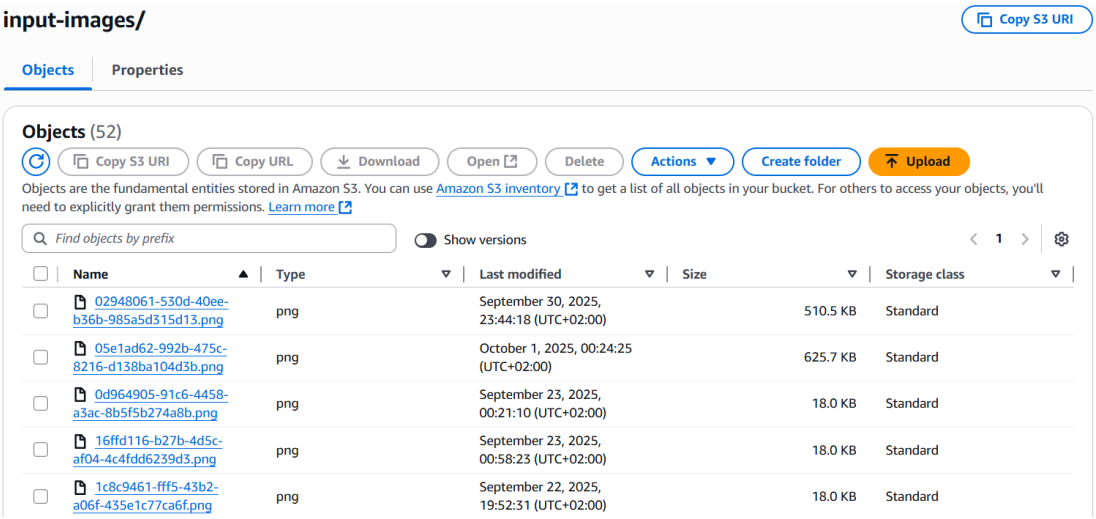


Figura 4.17: Carpeta de Amazon S3 utilizada para almacenar las imágenes de entrada subidas por los usuarios. En el último mes, se han procesado un total de 52 imágenes como parte de las pruebas de sistema.

4.4.3. Pruebas Unitarias

Las pruebas unitarias se han llevado a cabo para verificar el correcto funcionamiento de los distintos componentes de manera aislada, garantizando así la robustez y fiabilidad del sistema. Para su ejecución, se ha empleado la librería `tox` (3.1.6.1), que facilita la gestión de entornos virtuales y la automatización de pruebas mediante la integración con `pytest` (3.1.6.2). Además, `tox` permite generar informes de cobertura del código, proporcionando una métrica cuantitativa del grado de validación alcanzado.

Código 4.8: Gestión de los tests de la API con tox.

```
1 tox -e coverage
```

La ejecución de este comando genera un informe detallado con la cobertura global por fichero, permitiendo así identificar de forma precisa qué partes del sistema han sido verificadas y cuáles podrían requerir pruebas adicionales. La tabla 4.1 muestra el número de pruebas definidas por cada endpoint de los microservicios implementados.

Microservicio	Endpoint	Número de tests
API de Orquestación	/health	1
API de Orquestación	/authorize	4
API de Orquestación	/image-ai-analysis	8
API de Orquestación	/images/{service_name}/{image_id}	4
API de Denoising	/health	1
API de Denoising	/analyze-image	2
API de Inpainting	/health	1
API de Inpainting	/analyze-image	10
API de Upscaling	/health	1
API de Upscaling	/analyze-image	2
TOTAL		34

Tabla 4.1: Resumen de los tests realizados por endpoint en cada microservicio.

API de Orquestación de los modelos de IA

En la figura 4.18, se muestra el resultado de las pruebas realizadas sobre la API de orquestación. Se alcanzó un 88 % de cobertura global, cumpliendo así con el objetivo planteado. Puede observarse que la mayoría de los ficheros alcanzaron un 100 % de cobertura, mientras que el fichero `modelai_api_requests.py` solo llegó a un 17 %. Este comportamiento se debe a que dicho módulo incluye interacciones con las APIs de los modelos de IA que no fueron completamente simuladas en los tests.

```
================================================== 17 passed, 4 warnings in 1.45s ==================================================
coverage: commands[1]> coverage report -m
Name                                Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------------------
api\config\api_settigs.py              13      0      0      0   100%
api\routers\auth.py                    30      0      4      0   100%
api\routers\health.py                  10      0      0      0   100%
api\routers\orchestrator.py            54      0      0      0   100%
api\schemas\errors.py                  19      0      0      0   100%
api\schemas\orchestrator.py             9      0      0      0   100%
api\schemas\persistance.py             13      0      0      0   100%
api\schemas\status.py                   4      0      0      0   100%
api\utils\modelai_api_requests.py      29     23      6      0    17%   11-43
api\utils\orchestrator_service.py      54      0      4      0   100%
-------------------------------------------------------------------------------
TOTAL                                 235     23     14      0    88%
coverage: commands[2]> coverage xml -o C:\Users\34649\Documents\TFG_Informatica\aws-ai-orchestrator\backend\orchestrator-ai\.tox/coverage.xml
coverage: commands[3]> coverage html -d C:\Users\34649\Documents\TFG_Informatica\aws-ai-orchestrator\backend\orchestrator-ai\.tox/htmlcov
  coverage: OK (3.41=setup[0.27]+cmd[2.30,0.23,0.25,0.36] seconds)
  congratulations :) (3.69 seconds)
```

Figura 4.18: Cobertura de los tests unitarios realizados en la API de orquestación. Como se puede apreciar, a excepción del fichero `modelai_api_requests.py`, donde solo se logra una cobertura del 17 %, en el resto de ficheros principales de la API se logra cubrir el 100 % del código, alcanzando un 88 % de cobertura global de la API.

API de Denoising

La figura 4.19 refleja los resultados de las pruebas realizadas en la API de eliminación de ruido (Denoising). En este caso, la cobertura alcanzada fue del 89 %, destacando que la mayoría de los ficheros alcanzaron un 100 % de cobertura. El único caso en el que esto no ocurre es en el fichero `denoising_service.py`, donde la cobertura fue del 75 %. Este resultado confirma la validez del servicio principal y garantiza su correcto comportamiento en escenarios habituales.

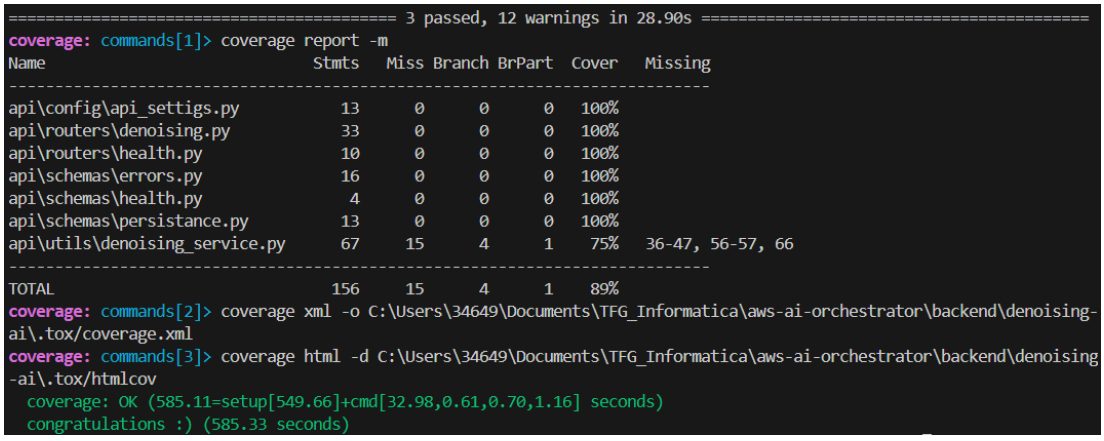


Figura 4.19: Cobertura de los tests unitarios realizados en la API de eliminación de ruido (Denoising). Como se puede apreciar, a excepción del fichero `denoising_service.py`, donde se logra una cobertura del 75 %, en el resto de ficheros principales de la API se logra cubrir el 100 % del código, alcanzando un 89 % de cobertura global de la API.

API de Inpainting

En el caso de la API de inpainting, que se encarga de la eliminación de texto superpuesto en imágenes, se obtuvo una cobertura global del 96 %. Como se muestra en la figura 4.20, casi todos los ficheros alcanzaron un 100 % de cobertura, a excepción de `inpainting_service.py`, que se situó en un 91 %. Estos resultados reflejan un nivel de validación muy elevado que asegura el correcto funcionamiento del servicio en las pruebas realizadas.

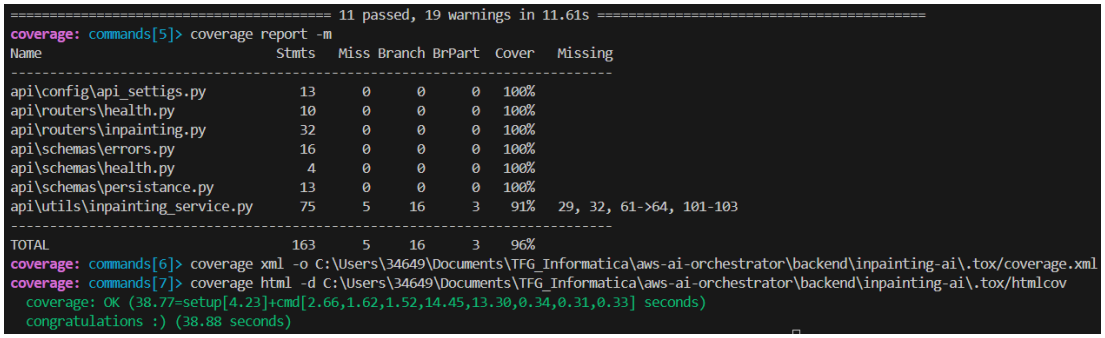


Figura 4.20: Cobertura de los tests unitarios realizados en la API de eliminación de texto superpuesto (Inpainting). En todos los ficheros principales de la API se logra cubrir el 100 % del código, a excepción de `inpainting_service.py`, donde se alcanza un 91 %. Globalmente se obtiene una cobertura del 96 % del código de la API.

API de Upscaling

Finalmente, en la API de aumento de resolución de imágenes (Upscaling), se alcanzó el mejor resultado global, con un 98 % de cobertura. La figura 4.21 muestra que prácticamente todos los ficheros cuentan con un 100 % de cobertura, salvo `upscaling_service.py`, que alcanzó un 92 %. Estos valores confirman que el servicio de upscaling se encuentra validado correctamente.

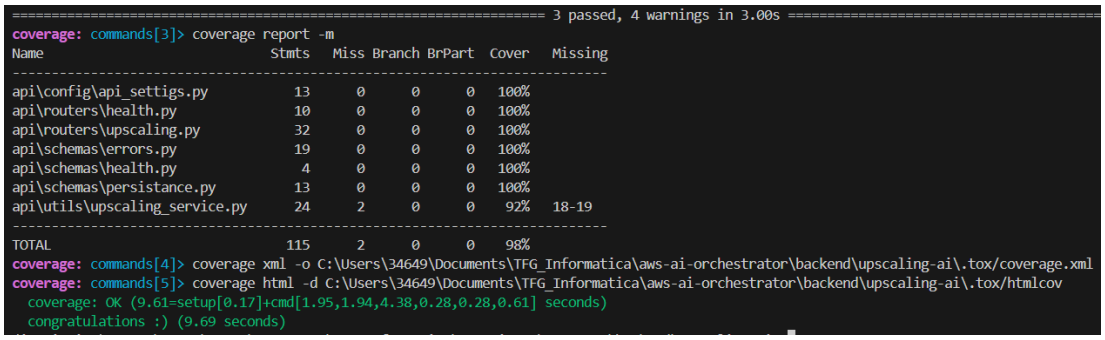


Figura 4.21: Cobertura de los tests unitarios realizados en la API de aumento de resolución (Upscaling). En todos los ficheros principales de la API se logra cubrir el 100 % del código, a excepción de `upscaling_service.py`, donde se alcanza un 92 %. Globalmente se obtiene una cobertura del 98 % del código de la API.

4.5. Distribución y despliegue

Dependiendo de la sección concreta de la aplicación a la que nos refiramos (infraestructura, backend o frontend), la forma de desplegar ha sido diferente.

4.5.1. Despliegue de la infraestructura con Terraform

La infraestructura ha sido configurada mediante código con **Terraform** (3.1.4) y se puede encontrar en la carpeta `iac/` en el repositorio de GitHub [43], accesible en el siguiente enlace: `https://github.com/ManuelGRT/aws-ai-orchestrator`.

Siguiendo la organización definida en el apéndice A, se ha creado el repositorio de Terraform siguiendo la misma estructura:

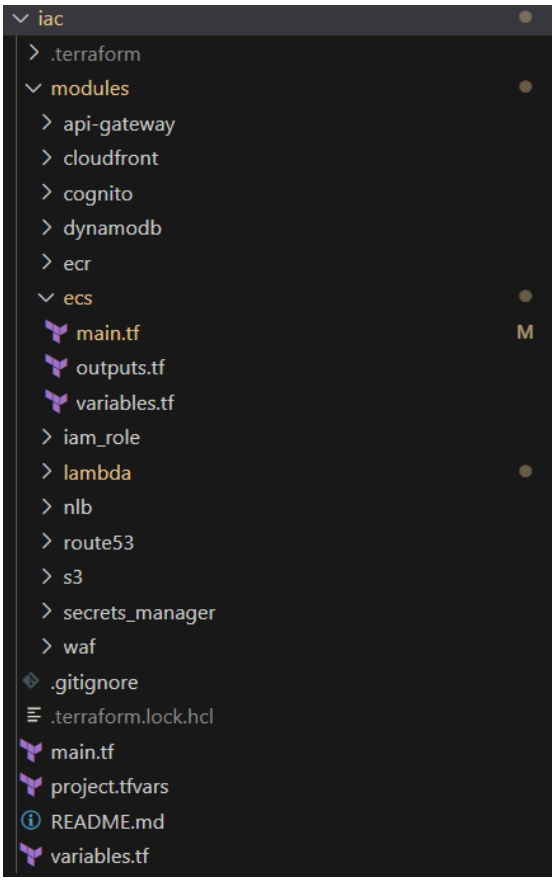


Figura 4.22: Estructura del repositorio de Terraform, siguiendo la metodología definida en el apéndice A.

Como se puede observar en la figura 4.22, cada una de las subcarpetas de `modules` corresponde a un recurso de AWS que se ha creado para este proyecto. Cada una de las subcarpetas contiene tres ficheros (`main`, `variables` y `outputs`).

Por ejemplo, a la hora de crear el Amazon S3 bucket de imágenes, tal como se refleja en el código 4.9, se definió, por un lado, el recurso principal (`aws_s3_bucket`) para crear el bucket de imágenes. A este se le añadió un "subrecurso", encargado de configurar la política del bucket, donde se especificó que no se puedan realizar accesos al bucket si no es una conexión https (`Deny Insecure Trnasport`).

Código 4.9: Amazon S3 Bucket de Imágenes

```
resource "aws_s3_bucket" "images_bucket" {
  bucket = var.s3_image_bucket_name
}

resource "aws_s3_bucket_policy" "images_bucket_policy" {
  bucket = aws_s3_bucket.images_bucket.id

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Sid       = "DenyInsecureTransport",
        Effect    = "Deny",
        Principal = {
          AWS = "*"
        },
        Action   = "s3:*",
        Resource = [
          aws_s3_bucket.images_bucket.arn,
          "${aws_s3_bucket.images_bucket.arn}/*"
        ],
        Condition = {
          Bool = {
            "aws:SecureTransport" = "false"
          }
        }
      }
    ]
  })
}
```

Configuración de Amazon Lambda con Terraform

La configuración del recurso de Amazon Lambda encargado de eliminar las imágenes del bucket de Amazon S3 pasado un mes se define en el IaC de la aplicación. Dentro de los módulos configurados con Terraform, en la sección `lambda`, se definió la siguiente estructura recogida en la figura 4.23.

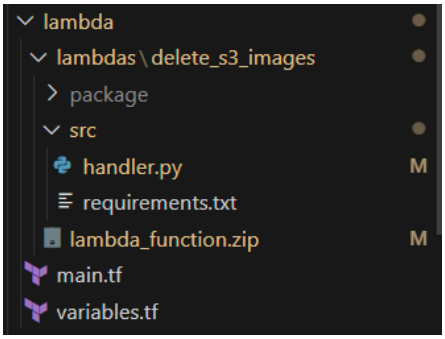


Figura 4.23: Estructura de configuración de la Amazon Lambda para eliminar imágenes del bucket de Amazon S3. Como se puede observar, la lambda está compuesta de dos ficheros fundamentales: `handler.py` y `requirements.txt`.

Como se puede observar, se definió una subcarpeta llamada `lambdas`, donde se configurarán las distintas Amazon Lambda que necesitemos para este proyecto. Para ello, dentro de la lambda que se desee configurar, se crearán dos ficheros: `handler.py` con el código de la Amazon Lambda, y `requirements.txt`, con los requisitos de las librerías que se necesitan para que el código funcione correctamente. Finalmente, para subir este código a AWS, en el código 4.10 se puede observar cómo se configura en el `main.tf` de Terraform un recurso para generar el .zip de la Amazon Lambda que después se subirá a AWS.

Código 4.10: Creación del .zip del recurso de Amazon Lambda

```
resource "null_resource" "resource_lambda_delete_s3_images" {
  triggers = {
    always_run = timestamp()
  }

  provisioner "local-exec" {
    interpreter = ["PowerShell", "-Command"]
    command = <<EOT
      New-Item -ItemType Directory -Force -Path
          "${path.module}/lambdas/delete_s3_images/package"
      pip install -r
          "${path.module}/lambdas/delete_s3_images/src/
                        requirements.txt" -t
          "${path.module}/lambdas/delete_s3_images/package"
      Copy-Item -Path
          "${path.module}/lambdas/delete_s3_images/src/*"
          -Destination
          "${path.module}/lambdas/delete_s3_images/package/"
          -Recurse -Force
    EOT
  }
}
```

Despliegue de la infraestructura del proyecto

El proceso de despliegue de la infraestructura en AWS mediante Terraform se realiza en dos fases principales. En primer lugar, se genera un plan de ejecución que permite validar los cambios antes de aplicarlos. Para ello se utiliza el siguiente comando:

Código 4.11: Generación del plan de ejecución con Terraform

```
terraform plan -var-file="project.tfvars"
```

Este comando analiza los archivos de configuración y compara el estado actual de la infraestructura en AWS con el estado deseado definido en el código. Como resultado, según se puede observar en la figura 4.24, se obtiene un plan detallado que muestra qué recursos serán creados, modificados o eliminados, lo cual permite revisar y confirmar que los cambios sean los esperados.

```
  # module.lambda.aws_lambda_function.lambda_delete_s3_images will be updated in-place
  ~ resource "aws_lambda_function" "lambda_delete_s3_images" {
        id                             = "lambda-delete-s3-images"
      ~ last_modified                  = "2025-08-26T09:18:09.000+0000" -> (known after apply)
      ~ source_code_hash               = "XQOXNLp4eyz94J8NUXm4g4cTNxcOs2SA6x6TklbhO3c=" -> (known after apply)
        tags                           = {}
        # (28 unchanged attributes hidden)

        # (4 unchanged blocks hidden)
    }

  # module.lambda.null_resource.resource_lambda_delete_s3_images must be replaced
-/+ resource "null_resource" "resource_lambda_delete_s3_images" {
      ~ id       = "2549820773775283121" -> (known after apply)
      ~ triggers = { # forces replacement
          ~ "always_run" = "2025-08-26T09:17:30Z" -> (known after apply)
        }
    }

Plan: 4 to add, 5 to change, 3 to destroy.
```

Figura 4.24: Ejemplo de `terraform plan`.

Posteriormente, una vez validado el plan, se procede a aplicar los cambios mediante el comando:

Código 4.12: Aplicación del plan de infraestructura en AWS

```
terraform apply -var-file="project.tfvars"
```

Con este paso, Terraform se encarga de crear y configurar automáticamente los recursos en AWS, asegurando que la infraestructura desplegada coincida con el estado definido en el código y gestionando las dependencias entre recursos.

4.5.2. Despliegue del frontend en Amazon S3

Para actualizar el frontend de la aplicación, en primer lugar es necesario generar un nuevo compilado del frontend. Este proceso se realiza ejecutando el siguiente comando:

Código 4.13: Generación del compilado del frontend de la aplicación

```
1 npm run build
```

Como resultado, se crea la carpeta `dist/`, que contiene la nueva versión del compilado del frontend. Después se sustituyen los archivos generados en dicha carpeta por los ya existentes en el bucket de Amazon S3 encargado de alojar los contenidos estáticos de la aplicación.

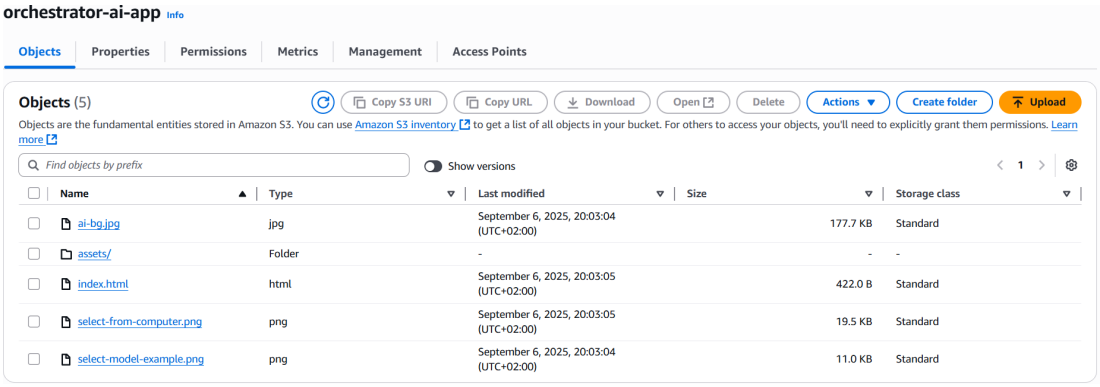


Figura 4.25: Contenido del bucket de Amazon S3 de la aplicación con el compilado del frontend

Una vez reemplazados los archivos en Amazon S3 (figura 4.25), es necesario invalidar la caché de Amazon CloudFront. Este paso, reflejado en la figura 4.26, garantiza que los usuarios reciban la versión más reciente de la aplicación. Con esta acción se concluye el proceso de despliegue del frontend.

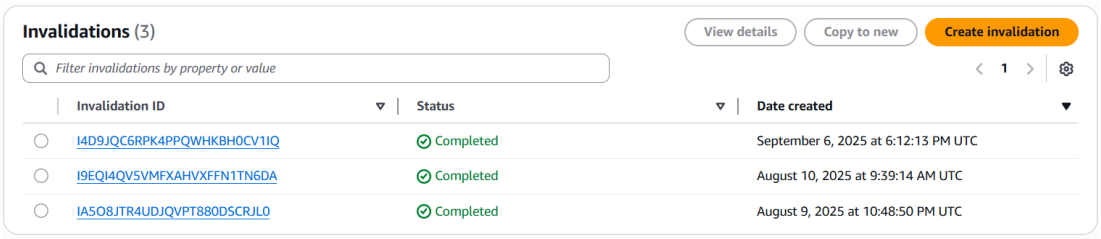


Figura 4.26: Invalidaciones de la caché realizadas en Amazon CloudFront para actualizar el frontend de la aplicación.

4.5.3. Despliegue del backend con Docker

Para el despliegue de cada microservicio, se utiliza un fichero **Makefile** (3.1.7.2), específico para cada API, en el que se recogen los principales comandos necesarios para la construcción y subida de las imágenes. Estos comandos engloban la autenticación en AWS, la creación de la imagen de **Docker** y la subida de dicha imagen a Amazon ECR.

```
aws-login:
    aws ecr get-login-password --region eu-west-1 | docker login --username AWS --password-stdin 435772683141.dkr.ecr.eu-west-1.amazonaws.com

docker-build:
    docker build -t orchestrator-ai-api-repository .

docker-tag:
    docker tag orchestrator-ai-api-repository:latest 435772683141.dkr.ecr.eu-west-1.amazonaws.com/orchestrator-ai-api-repository:latest

docker-push:
    docker push 435772683141.dkr.ecr.eu-west-1.amazonaws.com/orchestrator-ai-api-repository:latest
```

Figura 4.27: Fichero `Makefile` creado para el despliegue de la API de orquestación de los modelos de IA. Incluye los comandos necesarios para autenticarse en AWS, construir la imagen de Docker y subirla a Amazon ECR.

El uso de este fichero simplifica considerablemente el proceso de despliegue, ya que basta con ejecutar el comando correspondiente a través de una abreviatura definida en el propio Makefile, como se puede apreciar en la figura 4.27. Por ejemplo, para subir la imagen de Docker a Amazon ECR, bastaría con ejecutar:

Código 4.14: Subida de una imagen de Docker a Amazon ECR

```
1 make docker-push
```

De esta forma, se obtiene un proceso de despliegue más sencillo y menos propenso a errores. Como se muestra en la figura 4.28, Amazon ECR permite almacenar y gestionar distintas versiones de una misma API, lo que facilita el control de versiones y la trazabilidad de las imágenes subidas.

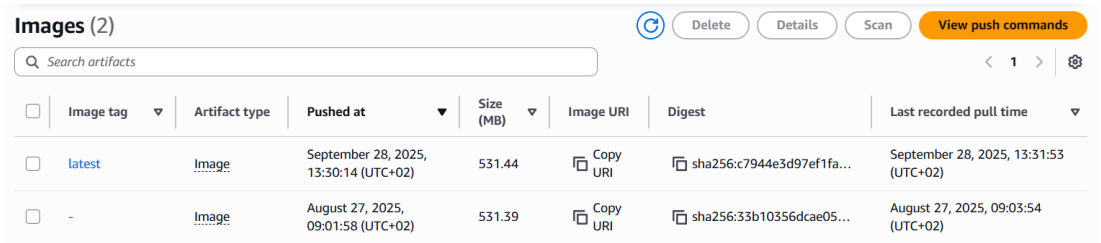


Figura 4.28: Imágenes de Docker subidas al repositorio de Amazon ECR asociado a la API de orquestación de modelos de IA.

Una vez publicada la imagen en Amazon ECR, el siguiente paso consiste en actualizar el servicio de Amazon ECS asociado. Para ello, se debe desplegar

nuevamente la tarea correspondiente, de manera que la API quede vinculada a la última versión de la imagen disponible en el repositorio. Con esto quedaría finalizado el despliegue del microservicio.

4.6. Análisis de costes de la infraestructura propuesta

Conforme se ha ido implementando el proyecto, el coste relativo al mantenimiento de la arquitectura ha ido escalando.

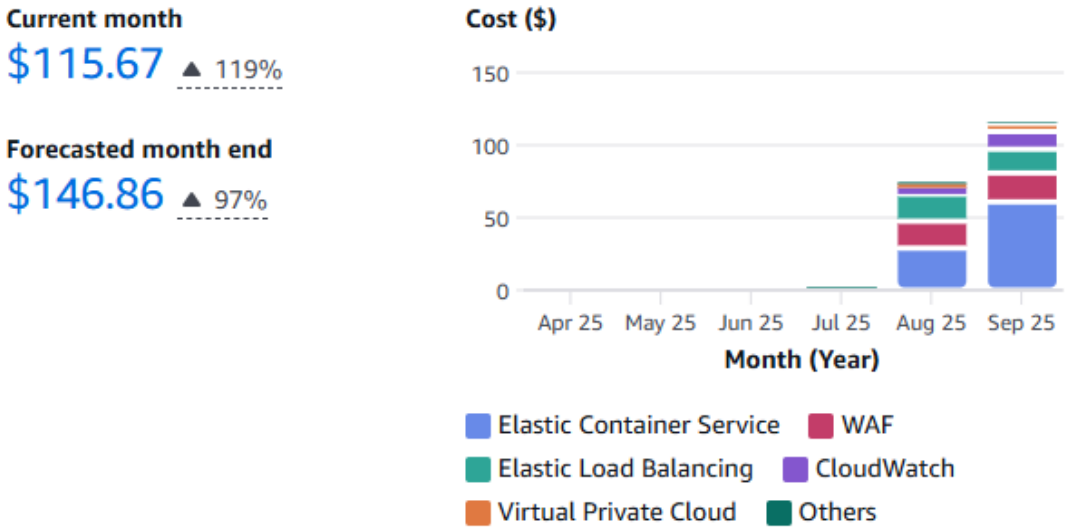


Figura 4.29: Representación del coste derivado del mantenimiento de la arquitectura de la aplicación en AWS. Como se puede observar, el coste ha ido escalando conforme se ha aumentado el número de recursos, con una previsión de que el precio ascenderá hasta los 150 euros. En concreto, el coste más elevado es el relativo a los recursos de Amazon ECS, seguido de los Amazon WAF para la API y el frontend de la aplicación, así como el Amazon NLB para controlar las llamadas a los microservicios.

Como se puede observar en la figura 4.29, el coste de la arquitectura ha ido creciendo cada mes, debido sobre todo al aumento de las capacidades de los recursos de Amazon ECS relativo a las APIs. Esto se debe a la alta capacidad de CPU y memoria, que se necesita para tener desplegadas APIs que gestionan modelos de inteligencia artificial.

Tras haber levantado toda la aplicación y también cada una de las APIs, y de haberla puesto en funcionamiento, el coste total de la aplicación se encuentra en torno a los **10 euros** diarios y, por tanto, en torno a **300 euros** al mes.

La optimización y reducción de costes es una tarea compleja, dado que el principal gasto de la infraestructura se concentra en las APIs de los microservicios desplegados en Amazon ECS. Estos microservicios se han configurado con los recursos mínimos de memoria (según el tamaño de su imagen de Docker) y CPU, necesarios para garantizar su correcto funcionamiento y despliegue. Por este motivo, las posibilidades de reducir significativamente el coste en este componente son limitadas.

Por lo tanto, la reducción de costes debería enfocarse en la optimización de otros servicios complementarios, como Amazon WAF o Amazon CloudWatch. Sin embargo, el impacto económico derivado de la optimización de estos recursos sería limitado.

5

Conclusiones y trabajos futuros

La parte final de este trabajo incluye, en la sección 5.1, una exposición de las conclusiones alcanzadas a partir de los objetivos planteados al inicio del proyecto (capítulo 2). Posteriormente, en la sección 5.2, se recogen una serie de propuestas y líneas de mejora que permitirían ampliar y optimizar la solución desarrollada. Finalmente, en la sección 5.3, se incluye una reflexión personal sobre la experiencia adquirida durante el desarrollo de este proyecto.

5.1. Conclusiones

El desarrollo de este trabajo ha permitido diseñar e implementar una arquitectura cloud para la orquestación de modelos de inteligencia artificial, desplegada íntegramente en Amazon Web Services (AWS). La solución propuesta ha sido validada mediante un caso de uso práctico enfocado en la restauración de imágenes, en el que se integran modelos de aumento de resolución, eliminación de ruido y eliminación de texto superpuesto. Los resultados obtenidos confirman que se ha conseguido alcanzar satisfactoriamente el objetivo principal establecido en el capítulo 2.

En relación con los objetivos específicos, se puede concluir lo siguiente:

- En la sección 4.2, se describe detalladamente la arquitectura necesaria para el despliegue y mantenimiento, tanto del frontend como del backend, cumpliendo con el objetivo 1. Se ha logrado una estructura modular y escalable,

que refleja de manera clara y comprensible cómo se comunican los distintos recursos de AWS.

- La definición y el despliegue de los recursos mediante infraestructura como código (IaC) se detallan en la sección 4.5.1, con lo que se alcanza el objetivo 2. El uso de Terraform ha permitido reproducir la infraestructura de forma automatizada, lo que garantiza mantenibilidad y trazabilidad.

- En cuanto al frontend (objetivo 3), su diseño e implementación se recogen en las secciones 4.3.1 y 4.5.2. Se ha desarrollado una interfaz intuitiva y accesible, desplegada mediante Amazon S3 y distribuida globalmente a través de Amazon CloudFront, asegurando con ello una experiencia de usuario fluida y de baja latencia.

- Respecto al backend (objetivo 4), se ha desarrollado una arquitectura basada en microservicios independientes implementados con FastAPI. Se han creado imágenes de las APIs mediante Docker, que se han desplegado en Amazon ECS, tal y como se detalla en la sección 4.3.2, cumpliendo también con el objetivo 5.

- Se ha garantizado la seguridad de la aplicación (objetivo 6), mediante la integración de Amazon Cognito, para la autenticación basada en tokens JWT, y de Amazon WAF, para la protección frente a ataques comunes. Ambos servicios se han configurado de manera coordinada, tanto en el frontend como en el backend.

- En relación con el almacenamiento y la gestión de datos (objetivo 7), se ha empleado Amazon S3 para almacenar las imágenes procesadas y Amazon DynamoDB para la persistencia de la información asociada a las APIs. Además, se ha implementado un proceso automatizado de limpieza, mediante una función Amazon Lambda con eventos programados con Amazon EventBridge, cumpliendo así el objetivo 8.

- Finalmente, el proceso de validación y pruebas (objetivo 9), descrito en la sección 4.4, ha permitido verificar el correcto funcionamiento del sistema. Se han realizado pruebas manuales de integración y de sistema. También se han realizado pruebas unitarias automatizadas del backend, obteniendo coberturas de código superiores al 85 % en los microservicios. Estas pruebas demuestran y confirman la fiabilidad y estabilidad de la arquitectura desarrollada.

El cumplimiento de todos los objetivos específicos y del objetivo general permite concluir que la arquitectura diseñada es sólida, escalable y segura. Además, el caso de uso desarrollado demuestra la viabilidad del sistema para la orquestación de modelos de inteligencia artificial en entornos cloud, lo que constituye una base sobre la que se pueden desarrollar futuros proyectos de mayor envergadura.

5.2. Trabajos Futuros

Los resultados obtenidos abren la puerta a posibles líneas de mejora y ampliación, que podrían abordarse en trabajos futuros. Una de las principales limitaciones encontradas durante el desarrollo de este TFG ha sido la capacidad de cómputo disponible en los entornos de ejecución. Debido a restricciones y limitaciones de coste, no ha sido posible desplegar instancias con recursos de GPU. Esto ha afectado a los tiempos de respuesta de algunos modelos, llegando en ocasiones a provocar errores de timeout.

Una de las posibles mejoras podría orientarse al uso de servicios específicos para el despliegue de modelos de inteligencia artificial, como **Amazon SageMaker** [45] [46]. Este servicio permitiría disponibilizar endpoints con instancias de GPU, lo que incrementaría considerablemente la eficiencia del sistema y reduciría la latencia percibida por el usuario. De forma alternativa, se podrían emplear recursos de **Amazon EC2** [47] [48] con instancias de GPU, configuradas para ejecutar los microservicios de IA en tiempo real.

Además, sería recomendable profundizar en la capa de persistencia, incorporando bases de datos relacionales como **Amazon RDS** [49] [50] para la gestión de datos estructurados, que podrían complementar el uso de Amazon DynamoDB según el tipo de información almacenada. Esta combinación híbrida mejoraría la flexibilidad y optimización del almacenamiento.

Otro elemento fundamental que debería incorporarse en una aplicación de este tipo es un sistema de alarmado que permita detectar y notificar posibles fallos en tiempo real. Para incluirlo, podría configurarse el servicio Amazon CloudWatch vinculándolo con **Amazon Simple Notification Service** (Amazon SNS) [51]. De este modo, cada vez que se produjera un error no deseado en alguno de los microservicios, el sistema enviaría automáticamente una alerta por correo electrónico, lo que facilitaría una respuesta rápida y minimizaría el impacto de las incidencias sobre la disponibilidad del servicio.

También se podría modificar la configuración de la conexión REST API, pasando del modo Regional a una distribución **Edge-Optimized** [52]. Esta modificación permitiría aprovechar la red global de Amazon CloudFront, y, por tanto, reduciría de forma significativa la latencia en la comunicación y mejoraría los tiempos de respuesta de la API de orquestación de los modelos de inteligencia artificial.

Por último, en el ámbito de la seguridad, una posible línea de mejora futura consistiría en implementar un sistema completo de gestión de usuarios mediante autenticación con credenciales en Amazon Cognito, ampliando el control de acceso.

5.3. Conclusiones Personales

Desde una perspectiva personal, el desarrollo de este proyecto me ha supuesto un reto y un aprendizaje. A pesar de contar inicialmente con un conocimiento no muy amplio sobre tecnologías cloud e inteligencia artificial aplicada, este trabajo me ha permitido adquirir una comprensión integral del proceso de diseño, despliegue y mantenimiento de aplicaciones en AWS que involucran modelos de inteligencia artificial.

A lo largo del proyecto, he podido alcanzar una visión global de la interacción entre los distintos servicios cloud, comprendiendo en profundidad conceptos como la infraestructura como código, la creación de imágenes con Docker y la orquestación de microservicios, entre otros.

En resumen, este Trabajo Fin de Grado ha sido una experiencia ilusionante y muy enriquecedora, que me ha permitido descubrir y manejar conceptos nuevos, y desarrollar competencias prácticas y teóricas en ámbitos clave de la ingeniería informática moderna, lo que me ha aportado una base sólida, para futuros proyectos profesionales o de investigación en el campo de la inteligencia artificial y la computación en la nube.

Bibliografía

[1] B. Eich and N. Communications, "Javascript, lenguaje de programación," https://developer.mozilla.org/en-US/docs/Web/JavaScript, 1995.

[2] E. You, "Vue.js, framework progresivo de javascript," https://vuejs.org/, 2014.

[3] P. S. Foundation, "Python, lenguaje de programación," https://www.python.org/, 1991.

[4] S. Ramírez, "Fastapi, framework web para python," https://fastapi.tiangolo.com/, 2018.

[5] Meta AI, "Pytorch," https://pytorch.org/, 2016.

[6] J. Tachella, M. Terris, S. Hurault, A. Wang, D. Chen, M.-H. Nguyen, M. Song, T. Davies, L. Davy, J. Dong, P. Escande, J. Hertrich, Z. Hu, T. I. Liaudat, N. Laurent, B. Levac, M. Massias, T. Moreau, T. Modrzyk, B. Monroy, S. Neumayer, J. Scanvic, F. Sarron, V. Sechaud, G. Schramm, R. Vo, and P. Weiss, "Deepinverse: A python package for solving imaging inverse problems with deep learning," *arXiv preprint arXiv:2505.20160*, 2025, versión preprint. [Online]. Available: https://arxiv.org/abs/2505.20160

[7] X. Wang, L. Xie, C. Dong, and Y. Shan, "Real-esrgan: Training real-world blind super-resolution with pure synthetic data," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, 2021. [Online]. Available: https://arxiv.org/abs/2107.10833

[8] B. Lai, B. Yu, Y. Du, S. Tang, Y. Ma, D. Yu, Y. Sun, B. Xia, Q. Dang, H. Zhao, X. Xiao, Y. Liu, Y. Ma *et al.*, "Pp-ocr: A practical ultra lightweight ocr system," *arXiv preprint arXiv:2009.09941*, 2020. [Online]. Available: https://arxiv.org/abs/2009.09941

[9] R. Suvorov, E. Logacheva, A. Mashikhin, A. Remizova, A. Ashukha, A. Silvestrov, N. Kong, H. Goka, K. H. Kim, J. Yoo, and V. Lempitsky, "Resolution-robust large mask inpainting with fourier convolutions," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2022. [Online]. Available: https://arxiv.org/abs/2109.07161

[10] HashiCorp, "Terraform," https://developer.hashicorp.com/terraform, 2014.

[11] Amazon Web Services, Inc., "What is infrastructure as code (iac)?" https://aws.amazon.com/what-is/iac/, 2025.

[12] F. Gessert, W. Wingerath, S. Friedrich, and N. Ritter, "Nosql database systems: a survey and decision guidance," *Computer Science Reports*, 2016, guía de decisiones para elegir sistemas NoSQL.

[13] Amazon Web Services, Inc., "Bases de datos nosql en aws," https://aws.amazon.com/es/nosql/, 2025.

[14] tox developers, "tox, automation and testing tool," https://tox.wiki/en/4.29.0/index.html, 2008.

[15] pytest developers, “pytest, testing framework for python,” https://docs.pytest.org/en/stable/, 2004.

[16] I. Docker, “Docker,” https://www.docker.com/, 2013.

[17] ——, “Dockerfile reference,” https://docs.docker.com/reference/dockerfile/, 2013.

[18] L. Yencken, “Makefile tutorial,” https://makefiletutorial.com/, 2018.

[19] Amazon Web Services, Inc., “Amazon web services,” https://aws.amazon.com/es/, 2025.

[20] ——, “Amazon cloudfront,” https://aws.amazon.com/es/cloudfront/, 2025.

[21] ——, “Amazon api gateway,” https://aws.amazon.com/es/api-gateway/, 2025.

[22] ——, “Amazon simple storage service (amazon s3),” https://aws.amazon.com/es/s3/, 2025.

[23] ——, “Aws waf - web application firewall,” https://aws.amazon.com/es/waf/, 2025.

[24] ——, “Amazon cognito,” https://docs.aws.amazon.com/cognito/, 2025.

[25] ——, *Amazon Cognito: Developer Guide*, https://docs.aws.amazon.com/cognito/latest/developerguide/, Amazon Web Services, 2025, copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

[26] ——, “Aws lambda,” https://aws.amazon.com/es/lambda/, 2025.

[27] ——, “Amazon eventbridge,” https://aws.amazon.com/es/eventbridge/, 2025.

[28] ——, “Elastic load balancing: Network load balancer,” https://aws.amazon.com/es/elasticloadbalancing/network-load-balancer/, 2025.

[29] ——, “Amazon elastic container registry (amazon ecr),” https://aws.amazon.com/es/ecr/, 2025.

[30] ——, “Amazon elastic container service (amazon ecs),” https://aws.amazon.com/es/ecs/, 2025.

[31] ——, “Amazon cloudwatch,” https://aws.amazon.com/es/cloudwatch/, 2025.

[32] ——, “Amazon dynamodb,” https://aws.amazon.com/es/dynamodb/, 2025.

[33] H. Hua, S. Sivasubramanian *et al.*, “Amazon dynamodb: A scalable, predictably performant, and fully managed nosql database service,” in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014.

[34] Amazon Web Services, Inc., “Aws identity and access management (iam) - manage roles,” https://aws.amazon.com/es/iam/features/manage-roles/, 2025.

[35] ——, “Amazon virtual private cloud (amazon vpc),” https://aws.amazon.com/es/vpc/, 2025.

[36] ——, “Amazon vpc security groups,” https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html, 2025.

[37] ——, “Amazon ec2 user guide - elastic network interfaces,” https://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/using-eni.html, 2025.

[38] Microsoft Corporation, “Visual studio code,” https://code.visualstudio.com/, 2015.

[39] Postman, Inc., “Postman,” https://www.postman.com/, 2012.

[40] S. F. Conservancy, “Git,” https://git-scm.com/, 2005.

[41] GitHub, Inc., “Github,” https://github.com/, 2008.

[42] Atlassian, "Comparing workflows: Gitflow workflow," https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow, 2025.

[43] M. G. R. Tirado, "AWS AI Orchestrator," https://github.com/ManuelGRT/aws-ai-orchestrator, 2025, repositorio en GitHub.

[44] K. Zhang, Y. Li, W. Ren, H. Yong, L. Kong, Q. Yan, X. Shen, and L. Zhang, "Plug-and-play image restoration with deep denoiser prior," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2021. [Online]. Available: https://arxiv.org/abs/2008.13751

[45] E. Liberty, F.-A. Bourassa, S. Abrol, R. Gilad-Bachrach, S. Moran, and A. Smola, "Elastic machine learning algorithms in amazon sagemaker," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 2202–2213, 2018.

[46] Amazon Web Services, Inc., "Amazon sagemaker," https://aws.amazon.com/es/sagemaker/, 2025.

[47] A. W. Services, "Amazon elastic compute cloud (ec2) – web service interface for resizable compute capacity in the cloud," 2006, whitepaper técnico de Amazon EC2.

[48] Amazon Web Services, Inc., "Amazon elastic compute cloud (amazon ec2)," https://aws.amazon.com/es/ec2/, 2025.

[49] ——, "Amazon relational database service (amazon rds)," https://aws.amazon.com/es/rds/, 2025.

[50] ——, "Amazon relational database service (rds): Best practices and architecture guide," Amazon Web Services, Tech. Rep., 2023, whitepaper técnico de AWS. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/rds-best-practices/

[51] ——, "Amazon simple notification service (amazon sns)," https://aws.amazon.com/es/sns/, 2025.

[52] ——, "Amazon api gateway - edge-optimized rest api," https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-edge-optimized-api.html, 2025.

[53] HashiCorp, "Standard module structure," https://developer.hashicorp.com/terraform/language/modules/develop/structure, 2025, Último acceso: septiembre 2025.

[54] M. G. R. Tirado, "Análisis e implementación de modelos de deconvolución para el procesamiento de imágenes digitales," Master's thesis, Universidad Rey Juan Carlos, 2025, trabajo Fin de Grado en Matemáticas.

[55] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2015, pp. 234–241.

Apéndices

A

Organización del repositorio de Terraform

Hemos seguido las recomendaciones oficiales de Terraform en cuanto a modularidad y estructura de repositorios, recogidos en su documentación [53], para asegurar una configuración clara, mantenible y escalable de los recursos. De acuerdo con la documentación oficial, hemos definido la estructura del repositorio de la siguiente manera:

```
├── modules
│   ├── api-gateway
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   ├── s3
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── vpc
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
├── main.tf
├── variables.tf
└── project.tfvars
```

Figura A.1: Estructura del repositorio de Terraform en módulos.

Como se refleja en la figura A.1, nuestra metodología aplicada consiste en:

- **Carpeta `modules/`**: contiene módulos auxiliares, cada uno en una subcarpeta específica para un recurso o componente (por ejemplo: Amazon S3, Amazon VPC, Amazon ECS, ...).
- En cada submódulo:
  - `main.tf`: define los recursos principales del módulo.
  - `variables.tf`: declara las variables de entrada configurables del módulo.
  - `outputs.tf`: especifica los recursos que el módulo exporta para su consumo externo.

  Estos nombres de archivo son los recomendados por HashiCorp para facilitar la comprensión y reutilización.
- En la raíz del repositorio hemos definido un `main.tf` principal, que gestiona la ejecución de los submódulos.
- **Archivo de variables por entorno**: los valores específicos del proyecto, como los nombres de los recursos, se establecen en un archivo `project.tfvars`, lo que permite separar la lógica de infraestructura de la configuración concreta del despliegue.

Esta organización permite que cada módulo sea autónomo, reusable y fácil de adaptar a distintos entornos o proyectos, cumpliendo así con los principios de infraestructura como código (IaC) establecidos por Terraform.

B

Inteligencia Artificial

B.1. Inteligencia Artificial

La **Inteligencia Artificial** (IA) es una rama de la Informática dedicada al diseño de sistemas capaces de simular comportamientos considerados inteligentes. Estos sistemas están orientados a resolver tareas que, hasta hace poco, requerían obligatoriamente la intervención humana, como, por ejemplo, la percepción visual, el razonamiento lógico, la planificación, el aprendizaje y la comprensión del lenguaje natural.

A diferencia de los métodos tradicionales, en los que el comportamiento está determinado por un conjunto de reglas lógicas y explícitas, los sistemas de inteligencia artificial modernos pueden aprender de forma autónoma, a partir de la experiencia y el entrenamiento. En las últimas décadas, esta área ha evolucionado hacia métodos basados en datos, en los que el **aprendizaje automático** ha pasado a ocupar un papel clave y central.

B.2. Aprendizaje Automático

El **aprendizaje automático** (machine learning) es una subdisciplina de la inteligencia artificial, cuyo objetivo es desarrollar algoritmos capaces de aprender patrones a partir de datos y de realizar tareas, como clasificación, regresión, clustering o generación de contenidos, sin necesidad de tener que ser programados específicamente para cada caso. En lugar de definir reglas, se entrena un modelo a partir de ejemplos, para que generalice y genere reglas propias y conclusiones que puedan aplicarse a datos nuevos.

Los algoritmos de aprendizaje automático tradicionales suelen exigir una importante labor de **análisis de características**; es decir, la extracción manual de variables relevantes del conjunto de datos. Aunque son eficaces y útiles en muchos escenarios, estos métodos presentan limitaciones cuando se trata de datos altamente complejos y estructurados, como, por ejemplo,

las imágenes o el lenguaje natural.

En este contexto, surge el **aprendizaje profundo** (deep learning), una evolución del aprendizaje automático.

B.3. Aprendizaje Profundo (Deep Learning)

El **deep learning** es una rama del aprendizaje automático que se basa en el uso de **redes neuronales artificiales profundas**; es decir, de redes formadas por múltiples capas que permiten modelar representaciones jerárquicas de los datos. A diferencia de los enfoques tradicionales, en los que las características se diseñan manualmente, en el deep learning la red aprende automáticamente las reglas y patrones más adecuados, directamente a partir de los datos no procesados.

Este paradigma resulta especialmente eficaz para resolver tareas de alta complejidad, en campos como el reconocimiento de imágenes, la traducción automática, el reconocimiento de voz o la restauración de señales degradadas. En concreto, en el ámbito del procesamiento de imágenes, el "aprendizaje profundo"ha superado de forma clara y decisiva a los métodos clásicos, y ha logrado resultados óptimos en clasificación, segmentación, detección de objetos y restauración de imágenes.

El éxito y la eficacia del deep learning han sido posibles gracias a tres pilares fundamentales: el incremento de la capacidad computacional (especialmente mediante GPUs), la disponibilidad de grandes volúmenes de datos y la mejora de las técnicas de entrenamiento.

El aprendizaje profundo proporciona el marco conceptual y técnico sobre el que se construyen arquitecturas como las **redes neuronales convolucionales**, las cuales permiten trabajar de manera eficiente y correcta con imágenes.

Antes de explicar en detalle las redes convolucionales, es necesario presentar tres conceptos fundamentales que constituyen la base de su funcionamiento: la **convolución**, como operación matemática sobre imágenes y las **funciones de activación**, que introducen no linealidad en el modelo.

B.3.1. Convolución

Es importante explicar el concepto de convolución, pieza clave en el proceso de transformación de una imagen, para entender cómo funcionan las redes neuronales convolucionales. Para ello, nos apoyamos en lo recogido en [54].

Definición. Dadas dos funciones f, $g : \Omega \to \Omega$, se define la **convolución continua** de f y g en t, denotada por $(f * g)(t)$, como:

$$(f * g)(t) = \int_{\Omega} f(x)g(t - x)\, dx \tag{B.1}$$

La operación de convolución puede entenderse como una forma de "mezclar" dos funciones. Esta operación permite modelar cómo una función f se ve afectada o modificada por un filtro o **núcleo de convolución** g, cuya forma determina el tipo de transformación que se aplica a f (por ejemplo, suavizado, realce de bordes, etc.)

En el plano discreto, la operación de convolución (B.1) se expresa de la siguiente manera:

$$(f * g)(x) = \sum_{i=-\infty}^{\infty} f(i)g(x-i) \tag{B.2}$$

Esta operación se aplica por separado a cada una de las capas de la imagen. Por lo tanto, la operación de convolución definida anteriormente estaría restringida a un dominio finito, ya que se trabaja con imágenes en 2D ($m \times n$), y el kernel (k) es una matriz de dimensiones $L \times L$, donde L es impar ($L = 2p+1$, $\forall p \in \mathbb{N}$). Generalmente, el tamaño de este kernel suele ser mucho menor que el de la imagen. La operación pasaría a representarse de la siguiente forma:

$$(k * f)(x, y) = \sum_{i=-p}^{p} \sum_{j=-p}^{p} k(i, j)f(x-i, y-j) \tag{B.3}$$

Esta fórmula sirve cuando fijamos como eje de coordenadas o eje para recorrer las matrices, la posición central del kernel y la posición central del pixel de la imagen que se va a convolucionar. Si queremos modificar este eje, haciendo que sea la posición de más arriba a la izquierda, tanto del kernel como de la imagen, de tal manera que el dominio en el que nos movemos pasaría a ser $i, j \in [0, k-1]$, habría que desplazar el sumatorio anterior obteniendo el siguiente resultado:

$$(k * f)(x, y) = \sum_{i=0}^{L-1} \sum_{j=0}^{L-1} k(i, j)f(x-i+p, y-j+p) \tag{B.4}$$

A continuación, se muestra un ejemplo sencillo para entender el proceso de convolución. Supongamos que nuestra imagen (f) es una matriz de 5x5 píxeles y el kernel que causa la convolucion es de dimensión 3x3 ($p = 1$):

$$f = \begin{bmatrix} f_{0,0} & f_{1,0} & f_{2,0} & f_{3,0} & f_{4,0} \\ f_{0,1} & f_{1,1} & f_{2,1} & f_{3,1} & f_{4,1} \\ f_{0,2} & f_{1,2} & f_{2,2} & f_{3,2} & f_{4,2} \\ f_{0,3} & f_{1,3} & f_{2,3} & f_{3,3} & f_{4,3} \\ f_{0,4} & f_{1,4} & f_{2,4} & f_{3,4} & f_{4,4} \end{bmatrix}, k = \begin{bmatrix} k_{0,0} & k_{1,0} & k_{2,0} \\ k_{0,1} & k_{1,1} & k_{2,1} \\ k_{0,2} & k_{1,2} & k_{2,2} \end{bmatrix}$$

La convolución en el píxel [2,2] ($f_{2,2}$) es de la siguiente forma:

$$(k * f)(2,2) = \sum_{i=0}^{2} \sum_{j=0}^{2} k(i, j)f(2-i+1, 2-j+1) = \sum_{i=0}^{2} \sum_{j=0}^{2} k(i, j)f(3-i, 3-j) =$$

$$= k_{0,0}f_{3,3} + k_{1,0}f_{2,3} + k_{2,0}f_{1,3} + k_{0,1}f_{3,2} + k_{1,1}f_{2,2} + k_{2,1}f_{1,2} + k_{2,2}f_{1,1}$$

Como se puede observar, la convolución transforma un píxel tomando únicamente los píxeles de alrededor. Dependiendo del tamaño del kernel, la "información" del entorno que usamos del píxel que estamos convolucionando puede ser mayor.

También es interesante notar que si invertimos tanto las columnas como las filas del kernel

de la siguiente forma:

$$k = \begin{bmatrix} k_{0,0} & k_{1,0} & k_{2,0} \\ k_{0,1} & k_{1,1} & k_{2,1} \\ k_{0,2} & k_{1,2} & k_{2,2} \end{bmatrix} \rightarrow k^* = \begin{bmatrix} k_{2,2} & k_{1,2} & k_{0,2} \\ k_{2,1} & k_{1,1} & k_{0,1} \\ k_{2,0} & k_{1,0} & k_{0,0} \end{bmatrix} \tag{B.5}$$

La operación de convolución pasaría a ser una multiplicación directa 1 a 1, si centramos el kernel en el píxel que queremos convolucionar:

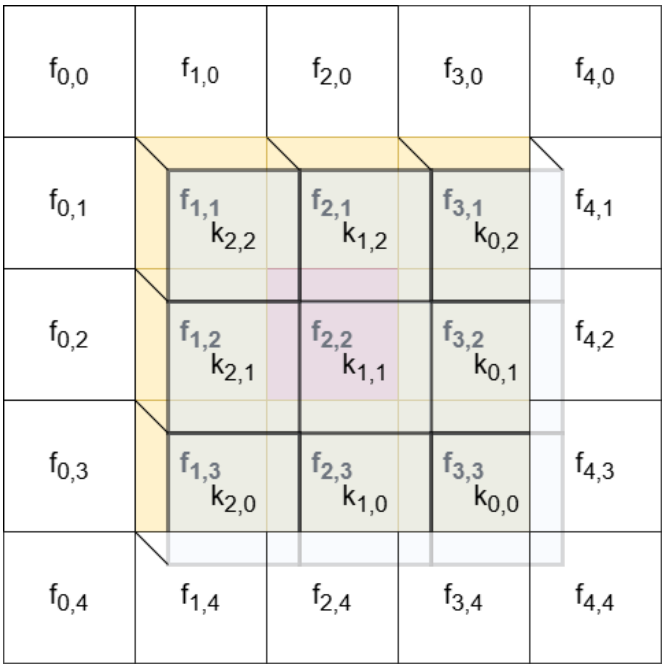


Figura B.1: Esquema visual de la relación directa entre la imagen y el kernel invertido, al aplicar la operación de convolución.

$$(k * f)(2,2) = \sum_{i=0}^{2}\sum_{j=0}^{2} k(i,j) f(3-i, 3-j) = \sum_{i=0}^{2}\sum_{j=0}^{2} k^*(i,j) f(i+1, j+1)$$

Lo que nos permite generalizarlo en la siguiente fórmula:

$$(k*f)(x,y) = \sum_{i=0}^{L-1}\sum_{j=0}^{L-1} k(i,j) f(x-i+p, y-j+p) = \sum_{i=0}^{L-1}\sum_{j=0}^{L-1} k^*(i,j) f(x+(i-p), y+(j-p)) \tag{B.6}$$

Para los píxeles localizados en el contorno de la imagen o en el borde de la misma, a la hora de calcular la operación de convolución se emplea la técnica denominada como **zero padding**. Esta técnica consiste en agregar ceros alrededor de una matriz (imagen), antes de aplicar la operación de convolución. El objetivo es añadir tantas filas y columnas de ceros como sean necesarias, para que el filtro pueda abarcar cada píxel sin salirse de los bordes. Es decir, el

número de filas y columnas viene dado por la siguiente fórmula:

$$\texttt{zero\_padding} = \frac{L-1}{2} \tag{B.7}$$

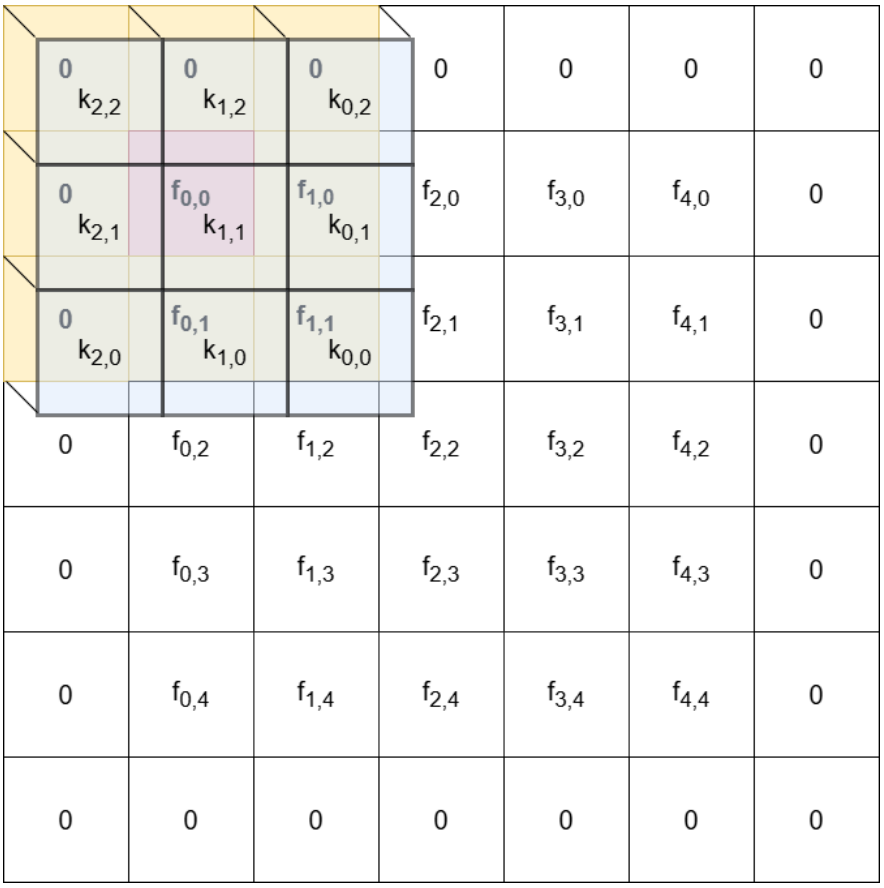


Figura B.2: Ejemplo de cálculo de la convolución en los píxeles de contorno, empleando la técnica de zero padding.

De esta forma, se logra poder realizar la convolución en todos los píxeles de la imagen. Este proceso se tendría que repetir píxel a píxel, hasta convolucionar todos los píxeles de la imagen. Sin embargo, al tratarse la convolución de una aplicación lineal, podemos expresar la convolución completa de todos los píxeles como el producto de una matriz A por un vector x que representa la imagen. Esta matriz es la que se conoce como la **Matriz de Toeplitz**.

Para ejemplificar cómo se crea esta aplicación lineal y, en particular, una matriz de Toeplitz, se parte de la imagen de dimensiones 5×5 que se ha definido previamente. Dado que el objetivo es calcular también la convolución en los píxeles del borde de la imagen, se añade un borde de ceros siguiendo la estrategia de zero padding, convirtiendo así la imagen en una matriz de

dimensiones 7×7:

$$f = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & f_{0,0} & f_{1,0} & f_{2,0} & f_{3,0} & f_{4,0} & 0 \\ 0 & f_{0,1} & f_{1,1} & f_{2,1} & f_{3,1} & f_{4,1} & 0 \\ 0 & f_{0,2} & f_{1,2} & f_{2,2} & f_{3,2} & f_{4,2} & 0 \\ 0 & f_{0,3} & f_{1,3} & f_{2,3} & f_{3,3} & f_{4,3} & 0 \\ 0 & f_{0,4} & f_{1,4} & f_{2,4} & f_{3,4} & f_{4,4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Vectorizamos la imagen anterior de la siguiente forma:

$$\begin{gathered} \mathbf{f} = (0, ..., 0, \\ 0, f_{0,0}, f_{0,1}, f_{0,2}, f_{0,3}, f_{0,4}, 0, \\ 0, f_{1,0}, f_{1,1}, f_{1,2}, f_{1,3}, f_{1,4}, 0, \\ 0, f_{2,0}, f_{2,1}, f_{2,2}, f_{2,3}, f_{2,4}, 0, \\ 0, f_{3,0}, f_{3,1}, f_{3,2}, f_{3,3}, f_{3,4}, 0, \\ 0, f_{4,0}, f_{4,1}, f_{4,2}, f_{4,3}, f_{4,4}, 0, \\ 0, ..., 0) \end{gathered}$$

Si también se toma de partida el mismo kernel (B.5) de tamaño 3×3, se define la matriz de Toeplitz (K) como:

$$K = \begin{bmatrix} k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ k_{0,2} & 0 & \cdots & 0 & k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 & 0 \\ k_{0,1} & k_{0,2} & 0 & \cdots & 0 & k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 & 0 \\ k_{0,0} & k_{0,1} & k_{0,2} & 0 & \cdots & 0 & k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 & 0 \\ 0 & k_{0,0} & k_{0,1} & k_{0,2} & 0 & \cdots & 0 & k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & k_{0,0} & k_{0,1} & k_{0,2} & 0 & \cdots & 0 & k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} & 0 \\ 0 & 0 & \cdots & 0 & k_{0,0} & k_{0,1} & k_{0,2} & 0 & \cdots & 0 & k_{1,0} & k_{1,1} & k_{1,2} & 0 & \cdots & 0 & k_{2,0} & k_{2,1} & k_{2,2} \end{bmatrix} \tag{B.8}$$

El número de ceros en cada columna que separa los distintos valores del kernel en la matriz es el número de ceros necesarios para llegar a M (en nuestro ejemplo, 7). Es decir, se rellena el kernel con el número de ceros necesarios, para que el kernel tenga las mismas filas que la imagen.

Si se generaliza, esta matriz da lugar a la siguiente aplicación lineal:

$$K^T\mathbf{f} = k * f_{i,j} \qquad \forall i \in [0, M-1],\, \forall j \in [0, N-1] \tag{B.9}$$

Por lo tanto, se puede describir la convolución de una imagen, como una aplicación lineal de la matriz de Toeplitz del kernel de convolución por la imagen vectorizada.

B.3.2. Funciones de activación

Las funciones de activación son un componente esencial en las redes neuronales artificiales, ya que permiten introducir **no linealidad** en el modelo. Sin ellas, una red compuesta únicamente por capas lineales (como combinaciones de productos matriciales y sumas) se comportaría como una transformación lineal, sin importar su profundidad. Esto limitaría su capacidad de modelar relaciones complejas en los datos. Al aplicar funciones de activación entre capas, la red adquiere la capacidad de aproximar funciones altamente no lineales, lo que resulta crucial en tareas como la clasificación, la segmentación o la restauración de imágenes.

En términos formales, tras calcular la combinación lineal de una entrada x con pesos w y sesgo b, es decir, $z = w^T x + b$, se aplica una función de activación ϕ que produce la salida de la neurona: $\phi(z)$.

B.3.3. Funciones de activación comunes

A continuación, se describen algunas de las funciones de activación más utilizadas en la práctica:

- **ReLU (Rectified Linear Unit)**:

$$\phi(z) = \text{máx}(0, z) \tag{B.10}$$

  Es una de las funciones más populares en redes profundas, debido a su simplicidad y eficiencia computacional. Introduce no linealidad descartando los valores negativos.

- **Leaky ReLU**:

$$\phi(z) = \begin{cases} z, & \text{si } z > 0 \\ \alpha z, & \text{si } z \leq 0 \end{cases} \tag{B.11}$$

  Variante de ReLU, que permite un pequeño gradiente negativo en la región $z < 0$.

- **Sigmoide**:

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{B.12}$$

  Transforma cualquier valor real en un rango entre 0 y 1. Aunque es útil para las tareas de clasificación binaria, puede presentar problemas de saturación en valores extremos, lo que afecta a la propagación del gradiente.

- **Tangente hiperbólica**:

$$\phi(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{B.13}$$

  Similar a la sigmoide, pero su rango está centrado en cero: $(-1, 1)$. También puede sufrir saturación.

- **Softmax**:

$$\phi(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{B.14}$$

  Se utiliza principalmente en la capa de salida para problemas de clasificación multiclase, ya que convierte el vector de salidas en una distribución de probabilidad.

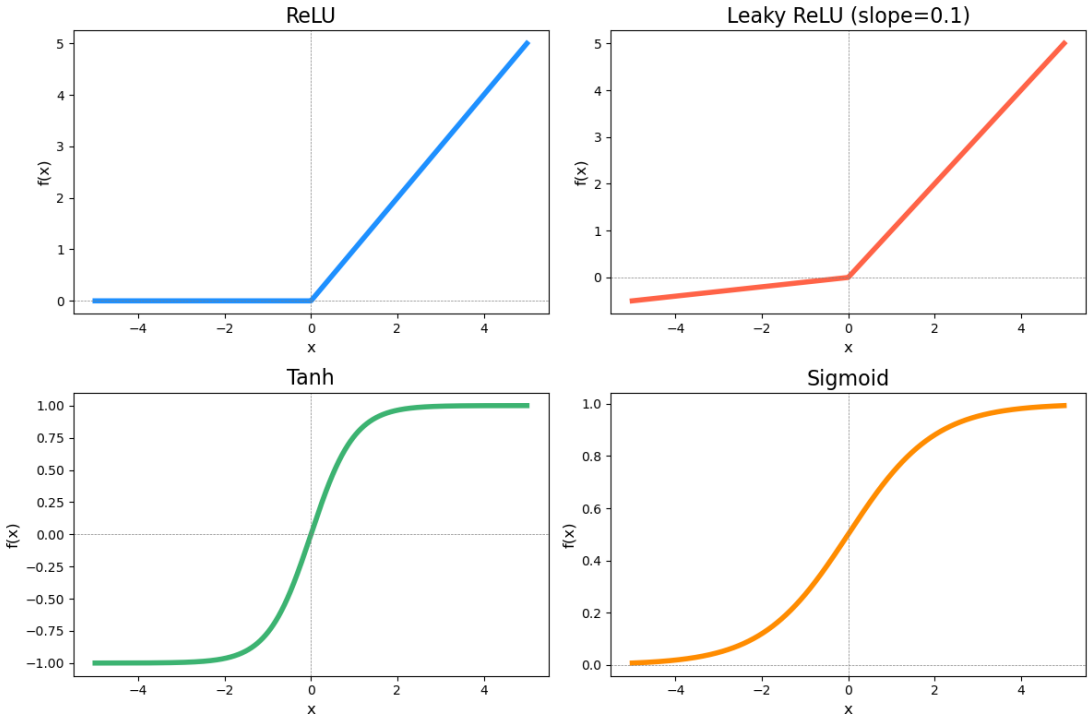


Figura B.3: Funciones de activación más comunes. Como se puede observar, ReLU y Leaky ReLU se comportan igual $\forall x \geq 0$.

Las funciones de activación desempeñan un papel clave en la generación de imágenes, ya que permiten a la red construir estructuras visuales complejas, a partir de entradas aleatorias. La elección adecuada de estas funciones es, por tanto, fundamental para la calidad de la imagen final.

B.4. Redes Neuronales Convolucionales (CNN)

Las **redes neuronales convolucionales** (CNN, por sus siglas en inglés: Convolutional Neural Networks) son un tipo especializado de redes neuronales profundas diseñadas específicamente para procesar datos con una estructura espacial regular, como las imágenes. A diferencia de las redes neuronales totalmente conectadas (**fully connected**), en las que cada neurona está conectada con todas las de la capa anterior, las CNN explotan la estructura local de los datos, mediante el uso de operaciones de convolución. Esto les permite aprender representaciones jerárquicas eficientes y con menos parámetros. Todo ello se puede apreciar en la figura B.4.

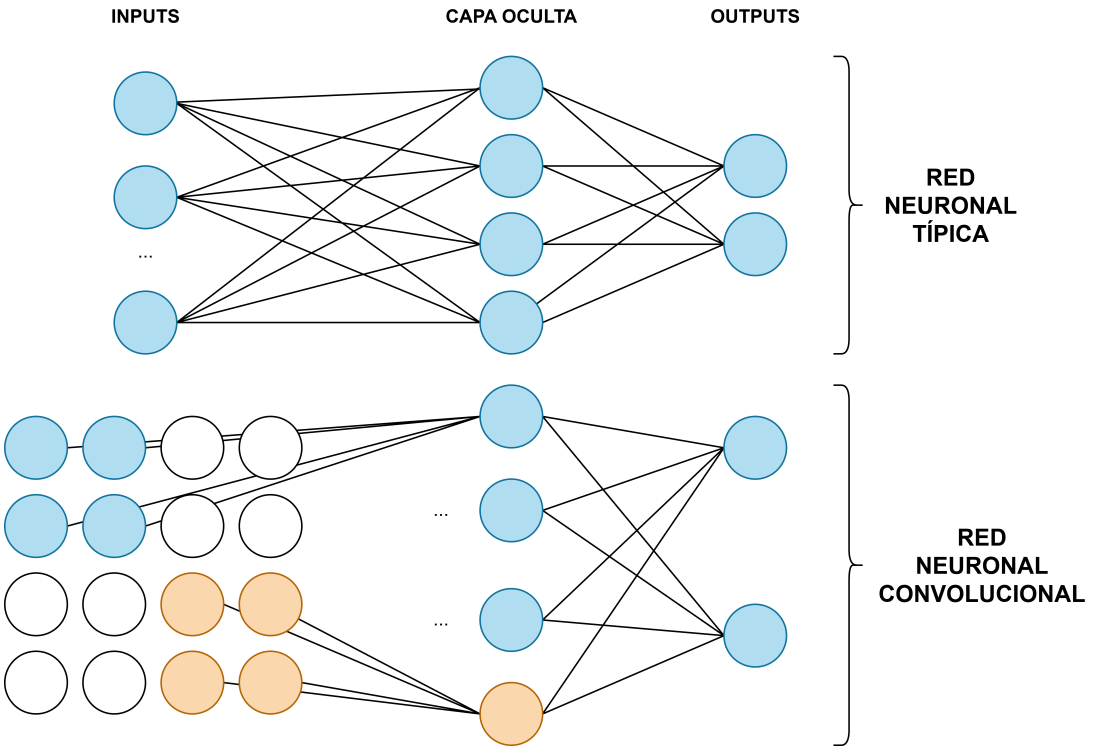


Figura B.4: Esquema de una red neuronal típica y una CNN. En las redes neuronales típicas, todas las neuronas están conectadas con todas las de la capa anterior. En las CNN, las neuronas de la capa oculta están conectadas solo con determinadas neuronas de la capa anterior.

En una imagen, los píxeles cercanos suelen estar relacionados. Las CNN aprovechan esta propiedad aplicando filtros (o kernels) que recorren la imagen, para detectar patrones locales. Estos filtros son matrices pequeñas de pesos entrenables que, al deslizarse sobre la imagen de entrada, generan nuevos mapas de activación donde se destacan ciertas características, como bordes, texturas o formas.

Una capa convolucional toma como entrada un conjunto de mapas (por ejemplo, una imagen RGB con tres canales), y aplica múltiples filtros para generar nuevos mapas de activación, aumentando por tanto el número de canales en las capas profundas. A medida que se avanza en la red, las capas convolucionales extraen características de mayor nivel de abstracción. Para ello, se aplica la siguiente fórmula:

$$K^T\mathbf{f} + B = k * f_{i,j} + b_{i,j} \qquad \forall i \in [0, M-1], \forall j \in [0, N-1] \tag{B.15}$$

Donde $K \in \mathbb{R}^{MN \times MN}$ es la matriz de Toeplitz del kernel $k \in \mathbb{R}^{L \times L}$, $\mathbf{f} \in [0,1]^{M \times N}$, la vectorización de la imagen f y B la matriz de sesgos.

Tal como se ha mencionado anteriormente, tras las capas convolucionales se aplican **funciones de activación**, como la ReLU, que introducen no linealidad en la red y permiten aprender relaciones complejas. Además, las CNN suelen incluir otras operaciones clave:

- **Pooling** o submuestreo. Reduce la resolución espacial de los mapas de activación, para

disminuir la carga computacional y hacer la red más robusta, frente a pequeñas variaciones en los datos.

- **Normalización**, como **batch normalization**. Acelera el entrenamiento y mejora la estabilidad.

La arquitectura general de una CNN se compone de varias capas convolucionales y de pooling en su fase inicial, seguida habitualmente de una o más capas totalmente conectadas (fully connected), para producir la salida final. Sin embargo, en tareas como la segmentación o la restauración de imágenes, en las que la salida también tiene forma de imagen, estas capas finales suelen ser reemplazadas por estructuras simétricas de expansión, como se verá en la arquitectura del siguiente apartado.

Gracias a su capacidad para aprender representaciones espaciales jerárquicas directamente desde los datos, las redes convolucionales se han convertido en la herramienta más eficaz para tareas de computer visión (visión artificial). En el contexto de este trabajo, la CNN constituye el bloque fundamental sobre el que se construyen arquitecturas más específicas, como **U-Net**.

B.5. U-Net

U-Net es una arquitectura perteneciente a las redes neuronales convolucionales, propuesta por Ronneberger et al. [55] en 2015, originalmente para tareas de segmentación de imágenes biomédicas. Desde su aparición, se ha convertido en una de las arquitecturas más influyentes y reutilizadas en el ámbito del procesamiento de imágenes mediante redes neuronales profundas. Su éxito y difusión se debe, en gran parte, a su capacidad para combinar de forma eficaz el contexto global de una imagen con detalles locales de alta resolución, lo que la hace especialmente útil en actividades donde es importante comprender la estructura general de la imagen, como preservar los bordes y texturas finas. Esta característica hace que U-Net se utilice con frecuencia en tareas de restauración de imágenes, como la eliminación de ruido, el desenfoque o la reconstrucción de regiones y zonas degradadas.

La arquitectura de U-Net recibe su nombre por su forma simétrica en "U" tal y como se ve en la figura B.5, ya que está compuesta por dos trayectorias principales: una ruta de contracción, también conocida como **encoder**, y una ruta de expansión o **decoder**. Estas dos partes están conectadas mediante enlaces horizontales, denominados **conexiones de salto** (**skip connections**), los cuales permiten transferir información de alta resolución desde el encoder al decoder en niveles de profundidad equivalentes, como se puede ver en la figura B.5.

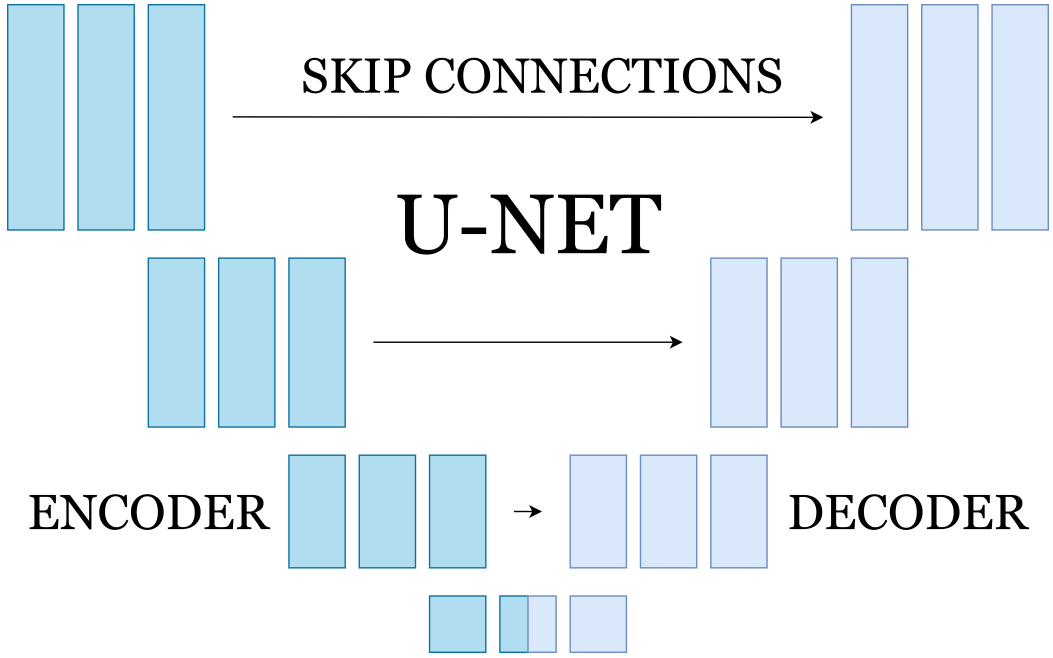


Figura B.5: U-Net con las secciones de Encoder, Decoder y Skip Connections

Una de las mayores ventajas de U-Net es que puede ser entrenada de manera eficiente, incluso con conjuntos de datos relativamente pequeños, ya que su diseño estructural facilita el flujo de gradientes y permite una generalización robusta. Además, al no depender de capas totalmente conectadas, la red es flexible respecto al tamaño de entrada y puede adaptarse fácilmente a diferentes resoluciones de imagen.

A continuación, se profundiza en cada una de las secciones en las que se divide la U-Net, para explicar en detalle el flujo que sigue la imagen a través de esta arquitectura.

B.5.1. Encoder

En la parte izquierda de la arquitectura, la ruta de contracción o **encoder** se encarga de capturar el contexto general de la imagen. Esta parte de la red tiene como objetivo extraer representaciones jerárquicas y profundas de la imagen de entrada.

Para ello, como se puede apreciar en la figura B.6, en cada capa convolucional se aplican una serie de convoluciones, con el objetivo de identificar diferentes rasgos distintivos de la imagen, cuyos filtros de convolución son, en general, de tamaño 3x3, seguidas de funciones de activación no lineales (normalmente ReLU, aunque podría ser otra). Al aplicarse valid-padding, después de cada convolución se reduce la dimensión de la imagen (se pierden los bordes de la imagen). Posteriormente, para pasar de una capa de convolución a otra, se aplican operaciones de reducción de dimensionalidad, como el **max pooling**.

En este tipo de arquitecturas, es común el uso de la función ReLU tras cada operación de convolución. Esta elección permite un entrenamiento más estable y eficiente, evitando el problema del desvanecimiento del gradiente, que afecta a funciones como la sigmoide o la tangente hiperbólica. Además, su implementación es muy sencilla y ha demostrado un rendimiento robusto en una gran variedad de tareas relacionadas con la visión artificial.

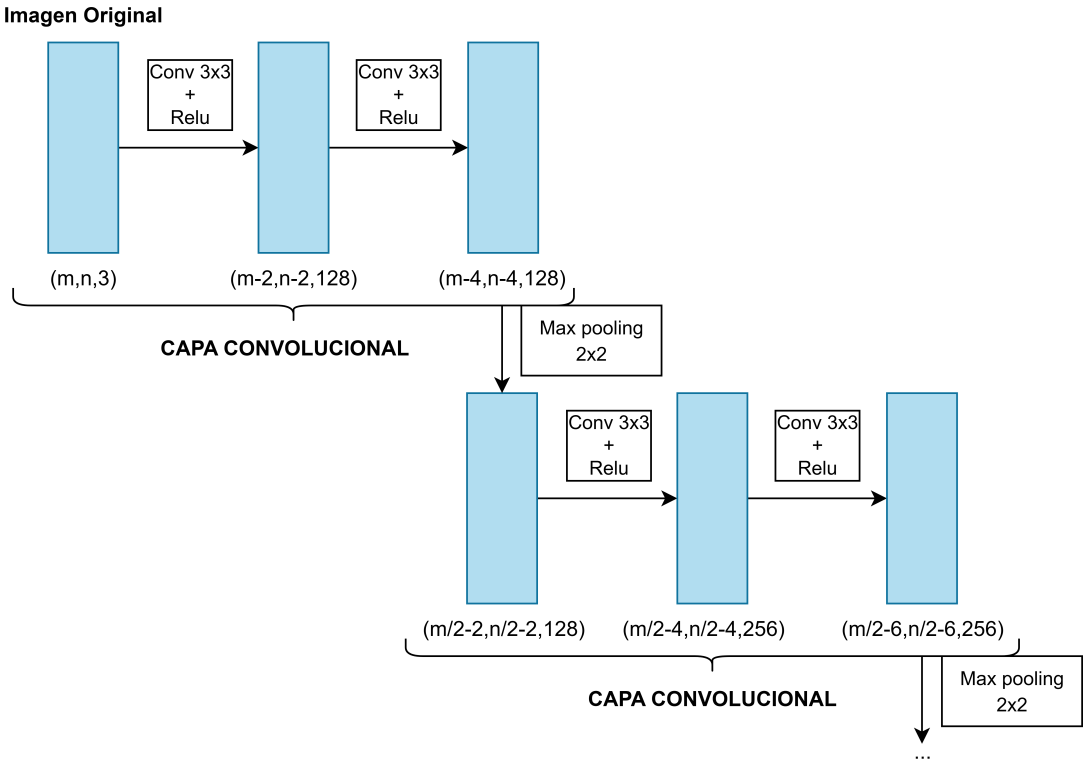


Figura B.6: Ejemplo de arquitectura de un encoder, para una imagen de dimensiones m $\times$ n $\times$ 3, tomando como 128 el número de canales iniciales de la primera capa convolucional

.

Con cada capa convolucional descendente, se reduce la resolución espacial de las características (altura y anchura), pero se aumenta el número de canales, lo que permite a la red representar estructuras de mayor complejidad semántica. Cada filtro en una convolución genera un nuevo canal. Por tanto, aumentan los canales, debido al aumento de filtros convolucionales.

A continuación, se explica detalladamente en qué consiste el max pooling.

Max Pooling

El **max pooling** es una operación común en redes convolucionales, que permite reducir la resolución espacial de las imágenes, a la vez que preserva las características más destacadas de cada región. Esta operación consiste en recorrer el mapa de activación con una ventana de tamaño fijo y seleccionar en cada región únicamente el valor máximo.

Por ejemplo, supongamos que tenemos una imagen f de entrada de tamaño 4×4:

$$\begin{bmatrix} f_{0,0} & f_{1,0} & f_{2,0} & f_{3,0} \\ f_{0,1} & f_{1,1} & f_{2,1} & f_{3,1} \\ f_{0,2} & f_{1,2} & f_{2,2} & f_{3,2} \\ f_{0,3} & f_{1,3} & f_{2,3} & f_{3,3} \end{bmatrix}$$

Si aplicamos un **max pooling** con una ventana de 2×2 y un **stride**, que define cuántas posiciones se desplaza la ventana cada vez que se aplica sobre la imagen, de 2, se selecciona el máximo valor en cada una de las siguientes subregiones:

$$\begin{bmatrix} f_{0,0} & f_{1,0} & f_{2,0} & f_{3,0} \\ f_{0,1} & f_{1,1} & f_{2,1} & f_{3,1} \\ f_{0,2} & f_{1,2} & f_{2,2} & f_{3,2} \\ f_{0,3} & f_{1,3} & f_{2,3} & f_{3,3} \end{bmatrix} \xrightarrow{\text{Max Pooling } 2\times2} \begin{bmatrix} max(f_{0,0}, f_{1,0}, f_{0,1}, f_{1,1}) & max(f_{2,0}, f_{3,0}, f_{2,1}, f_{3,1}) \\ max(f_{0,2}, f_{1,2}, f_{0,3}, f_{1,3}) & max(f_{2,2}, f_{3,2}, f_{2,3}, f_{3,3}) \end{bmatrix}$$

Como se puede apreciar, este proceso reduce la dimensión de la imagen (altura y anchura) a la mitad.

Además del max pooling, existen otros procesos de pooling diferentes, dependiendo del cálculo que se haga con los píxeles de al lado, como el **average pooling**, el cual, en vez de seleccionar el máximo, realiza una media de los píxeles involucrados.

B.5.2. Decoder y Skip Connections

La parte derecha, correspondiente a la ruta de expansión o **encoder**, tiene como finalidad reconstruir una imagen con la misma resolución que la original, a partir de las representaciones comprimidas del encoder. En este proceso, tal como se puede ver en la figura B.7, se aplican operaciones de **upsampling** o convoluciones transpuestas, para aumentar progresivamente la resolución espacial.

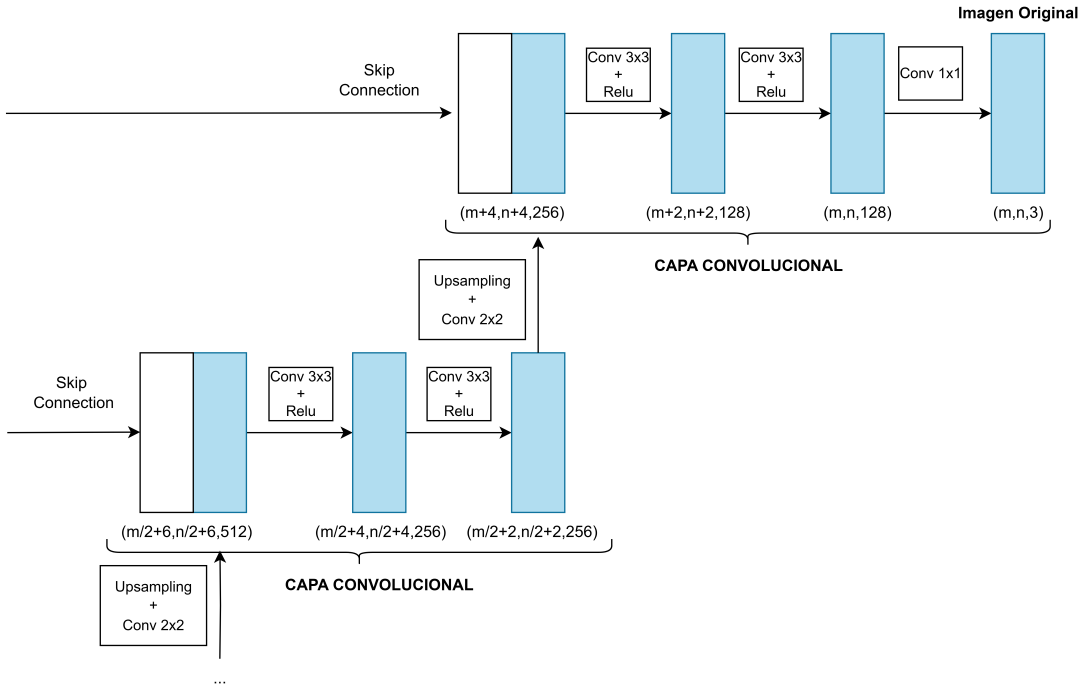


Figura B.7: Ejemplo de arquitectura de un decoder para una imagen de dimensiones m $\times$ n $\times$ 3

En cada etapa del decoder, la salida de la etapa anterior se enlaza con la correspondien-

te salida del encoder, mediante las **skip connections** (conexiones de salto). Esta operación permite recuperar información espacial detallada que pudo haberse perdido durante la reducción de resolución en el encoder. La combinación de representaciones profundas (del decoder) y características de baja frecuencia (del encoder) es esencial para que U-Net genere salidas coherentes y de alta calidad visual.

La última capa del decoder suele ser una convolución 1×1, que actúa como una proyección lineal para ajustar el número de canales al formato de salida deseado. Por ejemplo, en tareas de restauración de imágenes en escala de grises, se utiliza una salida de un solo canal, mientras que, en imágenes a color, se requieren tres canales (RGB).

Para reducir el número de canales en cada etapa del decoder, se utilizan capas convolucionales con un número de filtros convolucionales igual al número de canales de salida deseados. Es importante destacar que cada filtro convolucional tiene una **profundidad igual al número de canales de entrada**. Esto permite a cada filtro examinar de forma distinta la información proveniente de cada canal, y seleccionar de forma aprendida qué combinaciones de características son más relevantes para producir un nuevo canal de salida. Esta operación se muestra en la figura B.8.

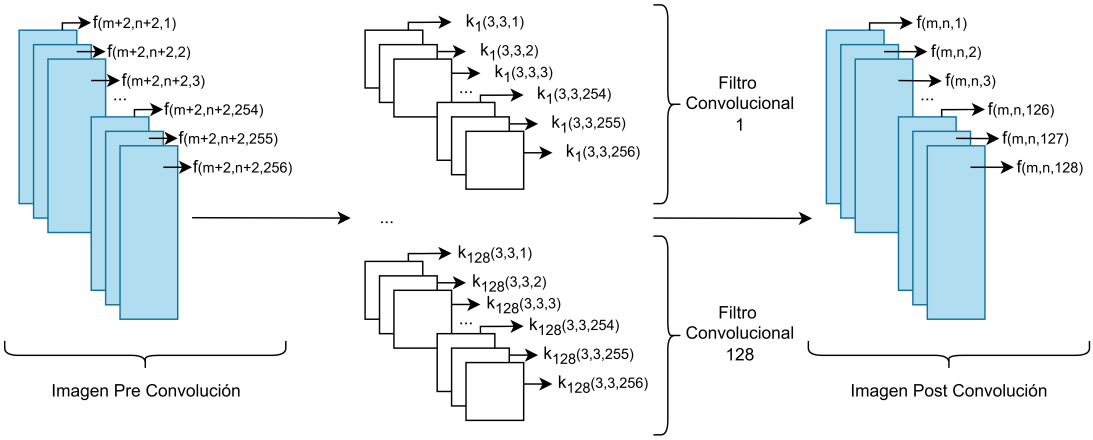


Figura B.8: Ejemplo de convolución de una imagen en la que se reduce el número de canales.

Como se aprecia en la figura B.8, cada filtro genera un canal de salida, a partir de la combinación ponderada de los canales de entrada. Así, para pasar de un tensor de dimensión $m \times n \times 256$ a uno de $m \times n \times 128$, se utilizan 128 filtros convolucionales de dimensiones $3 \times 3 \times 256$, donde 3×3 es el tamaño espacial del filtro, y cada filtro aprende a combinar selectivamente los 256 canales de entrada.

A continuación, se explicará el concepto de upsampling y cómo se implementa.

Upsampling

El **upsampling** (o "submuestreo inverso") es una operación cuyo objetivo es aumentar la resolución espacial (altura y anchura) de una imagen o mapa de activación. Esta técnica se utiliza comúnmente en redes convolucionales, para revertir la reducción de resolución que ocurre en las capas de pooling del encoder.

A diferencia del **max pooling**, que reduce la resolución para conservar solo la información más significativa, el upsampling intenta reconstruir una versión ampliada de la imagen, a partir de una representación más compacta.

Existen varias técnicas de upsampling. Una de las más utilizadas es el método de **Nearest Neighbor** (vecino más cercano). Este método consiste en replicar cada valor de píxel en una región más grande. Por ejemplo, si se parte de una imagen de tamaño 2×2, el proceso de upsampling mediante Nearest Neighbor sería el siguiente:

$$\begin{bmatrix} f_{0,0} & f_{1,0} \\ f_{0,1} & f_{1,1} \end{bmatrix} \xrightarrow{\text{Nearest Neighbor}} \begin{bmatrix} f_{0,0} & f_{0,0} & f_{1,0} & f_{1,0} \\ f_{0,0} & f_{0,0} & f_{1,0} & f_{1,0} \\ f_{0,1} & f_{0,1} & f_{1,1} & f_{1,1} \\ f_{0,1} & f_{0,1} & f_{1,1} & f_{1,1} \end{bmatrix}$$

De esta manera, se pasaría de una imagen 2×2 a una imagen 4×4, repitiendo simplemente los valores originales. Este procedimiento es computacionalmente eficiente, pero puede introducir saltos o bordes bruscos en la imagen resultante.

Por esta razón, en muchas arquitecturas, el upsampling suele ir seguido de una convolución, como se observa en la figura B.6 (en este caso, una convolución 2×2). Esta convolución posterior permite suavizar y refinar la imagen interpolada, favoreciendo transiciones más naturales y mejorando así la calidad visual del resultado.