# Agente Comercial de AI

**Tech Challenge – AI Engineer**

**Desarrollador:** Edgar Elias

**Email:** edgareliasc@outlook.com

**LinkedIn:** https://www.linkedin.com/in/edgar-elias-dev/

**GitHub:** https://github.com/edgarelias

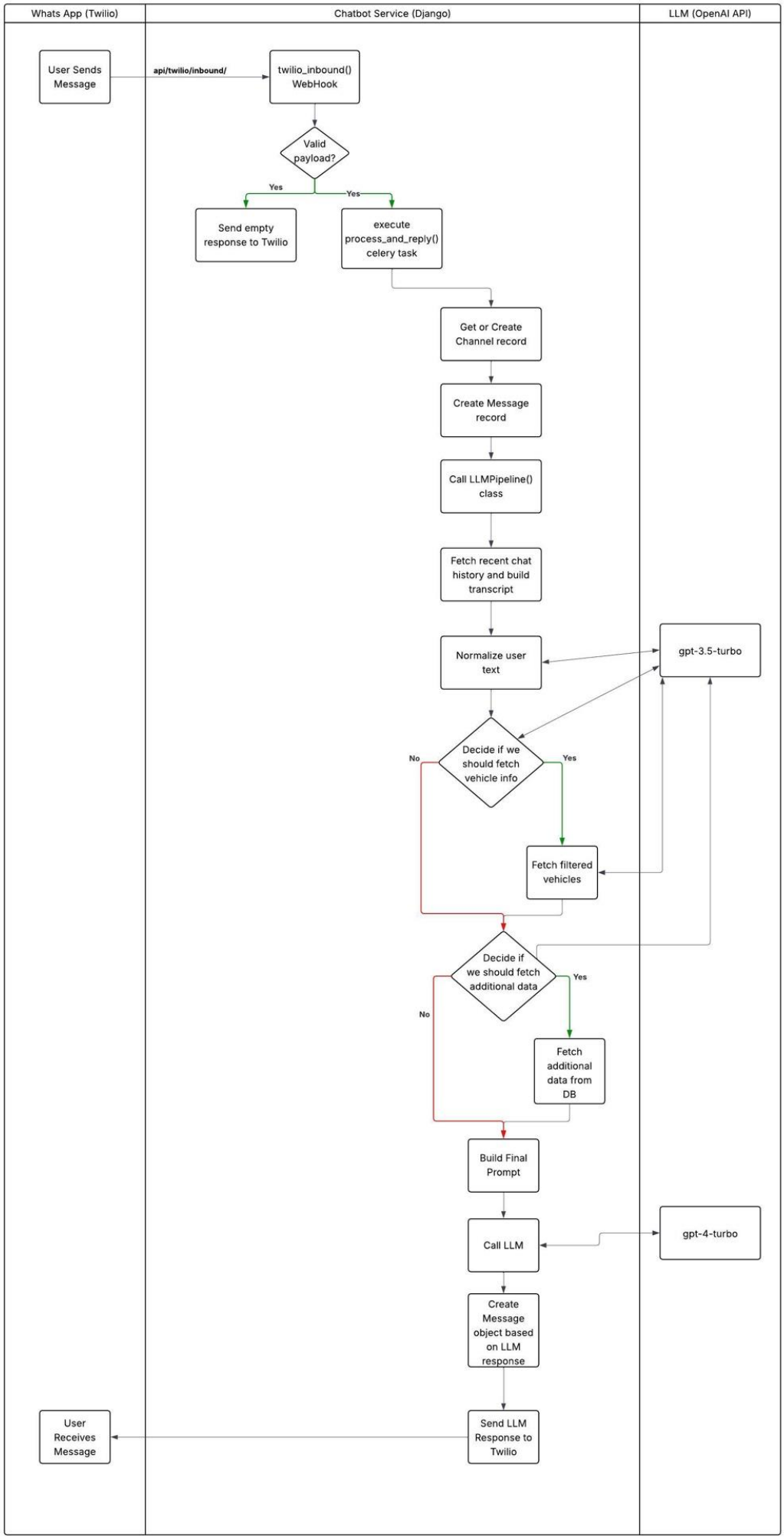
# Índice

# Introducción

El objetivo de este documento es presentar de manera clara la arquitectura, los endpoints, las clases, los diagramas, los prompts, las técnicas de RAG y el rendimiento de un agente de AI diseñado como agente comercial de ventas para Kavak.

# Arquitectura

1. Se utilizó la plataforma de Twilio para simular un bot de WhatsApp. Cuando un usuario envía un mensaje, Twilio realiza una petición POST al endpoint **api/twilio/inbound/.**
2. En la aplicación Django, la función **twilio_inbound** procesa y valida esa solicitud y, a continuación, desencadena la tarea asíncrona de Celery **process_and_reply**. Finalmente devuelve una respuesta vacía con estado HTTP 200.
3. La tarea **process_and_reply** obtiene o crea un registro de **Channel**, que almacena información sobre el número de WhatsApp que contacta la aplicación, y a continuación genera un registro de **Message** con el contenido enviado por el usuario. Para más detalles, consulte la sección de **Modelos**.
4. A continuación, se instancia la clase **LLMPipeline** y se invoca su método **process**, que coordina el flujo necesario para preparar la consulta al LLM y elaborar la respuesta. Véase la sección **LLM Pipeline** para más información.
5. Finalmente, la respuesta generada por el LLM se envía al usuario a través de la API de Twilio.


Whats App (Twilio)
Chatbot Service (Django)
LLM (OpenAI API)
User Sends Message
api/twilio/inbound/
twilio_inbound() WebHook
Valid payload?
Yes
Yes
Send empty response to Twilio
execute process_and_reply() celery task
Get or Create Channel record
Create Message record
Call LLMPipeline() class
Fetch recent chat history and build transcript
Normalize user text
gpt-3.5-turbo
Decide if we should fetch vehicle info
No
Yes
Fetch filtered vehicles
Decide if we should fetch additional data
Yes
No
Fetch additional data from DB
Build Final Prompt
Call LLM
gpt-4-turbo
Create Message object based on LLM response
Send LLM Response to Twilio
User Receives Message

# LLM Pipeline

El propósito de la clase LLM Pipeline es poder orquestar varios procesos antes de hacer una llamada al LLM para que pueda generarle una respuesta al usuario.

## Obtener historial de la conversación y generar un transcript

El primer paso del pipeline es obtener los últimos N (10) **Message** records que fueron agregados al **Channel** en los últimos N (15) minutos. Despues se va a generar un transcript que contenga los mensajes entre el agente (bot) y el ususario.

Se utiliza un timeframe de 15 minutos por default (puede ser configurado) porque si el usuario no ha mandado un mensaje en ese periodo de tiempo, el nuevo mensaje éntrate será tratado como una nueva interacción. Esto le da la habilidad de **multiturn** al agente (bot) y lo hace mas hábil.

Ejemplo del transcript:

*Usuario: Hola me gustaría comprar un Honda Civic*
*Bot: Encantado de ayudarte, que año lo estas buscando?*
*Usuario: 2012 en adelante*

## Normalizar el mensaje del usuario

Es muy común que el mensaje del usuario tenga errores ortográficos o gramáticos. Este paso del Pipeline limpia el input usando una llamada a gpt-3.5-turbo ya que es un modelo más rápido y se usa una temperatura de 0. Muchas empresas utilizan LLMs para limpiar el input de los usuarios con modelos livianos ya que dan mejores resultados que modelos de NLP.

**Prompt:**

*Corrige y normaliza la siguiente frase del usuario sobre autos. Devuelve el texto corregido en español, SIN añadir información adicional ni explicación. Ejemplo: 'nesesito un nissan versa 2022 en guadaljara' → 'Necesito un Nissan Versa 2022 en Guadalajara'.*

## Obtener informacion de los vehiculos

Ya que tenemos un Context Window limitado con los modelos proporcionados y un límite de 50,000 tokens por minuto, no queremos estar proporcionando toda la información de los vehículos en cada prompt. Si llegamos a incluir un inventario extenso en un prompt, nuestra llamada al LLM podría fallar.

Esto es una técnica de Retrieval-Augmented Generation (RAG) que combina búsqueda de información con generación de texto por un LLM.

Lo ideal sería tener un Vector Store (Elasticsearch, Pinecone, etc.) y almacenar toda la información de los vehículos. Al momento de la consulta del usuario podemos hacer una llamada al Vector Store con el mensaje del usuario y obtener los vehículos relacionados a su consulta sin tener que llamar al LLM.

En este paso del Pipeline se hacen dos consultas. Primero, se hace una consulta al LLM para checar si debemos de buscar y filtrar vehículos. En este prompt se pasa el transcript de los últimos N mensajes de la conversación al igual que el mensaje que el usuario acaba de mandar. Esto nos permite darle más contexto LLM de su trabajo y sus siguientes pasos. La información del vehículo ya puede estar en el transcript, lo cual nos puede ahorrar un paso. Esto también nos ayuda con el **multiturn** en caso de que la conversación haya cambiado de contexto (context switching).

Segundo, si necesitamos filtrar más vehículos, se hace una llamada al LLM para que retorne un CSV de los vehículos relacionados con la consulta del usuario. Se utiliza gpt-3.5-turbo ya que es un modelo más rápido y con una temperatura de 0.

**Prompt para checar si debemos consultar vehículos:**

*Eres un agente de ventas de autos de Kavak. Analiza la siguiente conversación y determina si el usuario ha solicitado información de vehículos, marcas de vehículos, modelos, kilometraje o rango de precios. Este paso es importante para decidir si se debe buscar información adicional.*

*Si es así, responde 'true'; de lo contrario 'false'.*

*Conversación:*

*{transcript}*

*Último mensaje del usuario:*

*{last_user_message}*

**Prompt para filtrar vehiculos:**

*Eres un agente de ventas de autos de Kavak. Se te proporciona un CSV con datos de vehículos. Filtra los vehículos según la siguiente consulta del usuario y devuelve un CSV con los que cumplan con la consulta del usuario. No inventes nada. Si no hay coincidencias, devuelve un CSV con solo el encabezado.*

*Consulta:*

*{user_msg}*

*CSV de vehículos:*

*{vehicles_csv}*

*Formato de salida:*

*stock_id,km,price,make,model,year,version,bluetooth,largo,ancho,altura,car_play*

## Obtener artículos de conocimiento relevantes

El challenge nos pide que el bot debe *Responder información básica sobre la propuesta de valor de Kavak*. En este bot se creó un modelo llamado **KnowledgeArticle**. El propósito de este modelo es almacenar información relevante al bot que tengan que ver con Kavak como políticas de la empresa, términos de privacidad y condiciones, temas de facturación, propuesta de valor, etc. El usuario puede almacenar texto relevante de forma manual o proveer un URL para que el bot pueda descargar su texto de una automáticamente.

Uno de los pasos de pipeline es checar si necesitamos cargar datos de este modelo antes de llamar al prompt final. Esta es otra técnica de RAG. El prompt le va a proporcionar los títulos de los **KnowledgeArticle** presentes en la base de datos y retornar una lista de los artículos relevantes al mensaje del usuario. Después, se obtendrán los **KnowledgeArticle** señalados por LLM del base de datos (PostgreSQL). Se utiliza gpt-3.5-turbo ya que es un modelo más rápido y una temperatura de 0.

**Prompt:**

*Eres un agente de ventas de autos de Kavak. A continuación, tienes una lista de artículos de conocimiento en formato id: título:*

*{id_block}*

*Basándote en la conversación previa y en la última pregunta del usuario, devuélveme un JSON con un arreglo de los IDs (UUID) de los artículos que sean relevantes. Si ninguno aplica, devuelve [].*

*Conversación previa:*

*{transcript}*

*Último mensaje del usuario:*

*{last_user_message}*

*Ejemplo de salida: ["uuid1", "uuid2"]*

## Prompt Final

En este paso se va a construir el prompt final para poder responder la pregunta del usuario. El prompt va a tener contexto de los mensajes previos (transcript), la última consulta del usuario, la lista de vehículos filtrada, y los artículos de conocimiento relevantes. Estos datos le dan suficiente contexto al LLM para poder dar una respuesta mas certera y no alucinar al momento de dar una respuesta. A diferencia de las otras llamadas, se utiliza gpt-4-turbo ya que es un modelo con mayor razonamiento y una temperatura de 0.9 para poder dar respuestas más creativas al usuario.

**Prompt:**

*Eres un agente de ventas de autos de Kavak. Responde solo usando la información proporcionada. No inventes autos, características ni promociones.*

*También debes tomar en cuenta crear planes de financiamiento para los autos que el usuario solicite, tomando como base el enganche y el precio del auto. La tasa de interés es del 10% y el plazo es de 3 a 6 años. No puedes considerar un plazo más largo ni una tasa de interés menor.*

*Si el CSV de vehículos no contiene información del vehículo que el usuario busca, responde con un mensaje amable al usuario diciéndole que el auto que busca no está disponible. Pregúntale si tiene otro vehículo en mente para que lo puedas buscar.*

*Para formatear tu respuesta en WhatsApp, utiliza el siguiente markdown:*

**texto* para negritas*

*_texto_ para itálicas*

*El formato de salida es un mensaje de WhatsApp en español, sin etiquetas HTML ni encabezados, y sin emojis ni abreviaciones. No agregues markdown que no esté especificado.*

*Último mensaje del usuario:*

*{last_user_message}*

*Conversación previa:*

*{transcript}*

*Vehículos filtrados (CSV):*

*{vehicle_section}*

*Información adicional:*

*{knowledge_articles}*

# Modelos

## BaseModel

| Campo | Type | Descripcion |
|---|---|---|
| date_created | DateTimeField | Marca de tiempo de creación |
| date_updated | DateTimeField | Marca de tiempo de última actualización |

## Channel

| Campo | Type | Descripcion |
|---|---|---|
| id | UUIDField | Identificador único |
| external_id | CharField | Identificador externo (único) |
| date_created | DateTimeField | Marca de tiempo de creación |
| date_updated | DateTimeField | Marca de tiempo de última actualización |

## Message

| Campo | Type | Descripcion |
|---|---|---|
| id | UUIDField | Identificador único |
| channel | ForeignKey | Canal al que pertenece este mensaje |
| text | TextField | Contenido del mensaje |
| author | CharField | Autor del mensaje (opcional) |
| date_created | DateTimeField | Marca de tiempo de creación |
| date_updated | DateTimeField | Marca de tiempo de última actualización |

## Vehicle

| Campo | Type | Descripcion |
|---|---|---|
| id | UUIDField | Identificador único |
| stock_id | CharField | Identificador de stock (único) |
| km | PositiveIntegerField | Kilometraje del vehículo (kilómetros) |
| price | FloatField | Precio del vehículo |
| make | CharField | Fabricante del vehículo |
| model | CharField | Modelo del vehículo |

| | | |
|---|---|---|
| year | PositiveIntegerField | Año de fabricación |
| version | CharField | Versión del vehículo (opcional) |
| bluetooth | BooleanField | Disponibilidad de Bluetooth |
| car_play | BooleanField | Disponibilidad de CarPlay |
| largo | FloatField | Longitud del vehículo |
| ancho | FloatField | Ancho del vehículo |
| altura | FloatField | Altura del vehículo |

## KnowledgeArticle

| **Campo** | **Type** | **Descripcion** |
|---|---|---|
| id | UUIDField | Identificador único |
| name | CharField | Título del artículo |
| text | TextField | Contenido del artículo (opcional) |
| url | URLField | URL de contenido adicional (opcional) |
| active | BooleanField | Estado del artículo (activo/inactivo) |
| date_created | DateTimeField | Marca de tiempo de creación |
| date_updated | DateTimeField | Marca de tiempo de última actualización |

# API Endpoints

| **Endpoint** | **Operaciones** | **Descripción** |
|---|---|---|
| /api/authentication/login/ | C | Autentica al usuario con el servicio de Django. |
| /api/chat/channels/ | CRUD | Hacer CRUD al modelo Channel |
| /api/chat/messages/ | CRUD | Hacer CRUD al model Message |
| /api/chat/channels/<id>/messages | R | Visualizar los mensajes relacionados con ese Channel |
| /api/catalog/vehicles/ | CRUD | Hacer CRUD al modelo Vehicle |
| /api/catalog/vehicles/import-csv/ | C | Te permite cargar un archivo CSV para actualizar el catalogo de Vehicle. Se comprueba si un Vehicle ya existe mediante el Stock ID. En case de existir, se actualizan los datos. |
| /api/catalog/knowledge_articles/ | CRUD | Hacer CRUD al modelo KnowledgeArticle. En caso de un créate o update y el modelo tenga el field URL. Se actualiza el contenido del text de forma automática y asíncrona. |
| /api/credentials_check | R | Valida que las integraciones y credenciales sean validas. |

# Tests

## Desempeño del agente

### Métricas Técnicas

Para garantizar un servicio robusto, mediremos la latencia de respuesta —el tiempo desde la petición hasta la respuesta— con un objetivo de menos de 3 segundos en el 80 % de los casos. Esto se puede medir mediante logs y la escabilidad de celery workers.

### Calidad de respuesta

Para minimizar las alucinaciones, mediremos el porcentaje de respuestas que contienen datos inventado. También verificaremos la cobertura de contenido, asegurando que al menos el 90 % de las preguntas sobre catálogo, financiamiento, FAQs y políticas obtengan fragmentos relevantes.

### Impacto de Negocio

Para valorar el retorno tangible, calcularemos la tasa de conversión de chats a redirects válidos. En algun punto el bot podría llegar a hacer Langchain para mandar emails con cotización o automatizar el apartado de vehículos mediante los APIs de Kavak. Con nuestro modelo de Channel, podriamos que margen de usuarios adquieron un producto de Kavak mediante el bot.

## Prevenir retroceso en su funcionalidad

### Suite de pruebas automatizadas

Mantener un conjunto de pruebas unitarias y de integración que cubre cada parte importante: normalización de datos, fases de RAG, generación de prompts y lógica de respuesta. Cada vez que se actualiza el código, estas pruebas se ejecutan para asegurar que todo siga funcionando igual.

### Pruebas de regresión

Se definen ejemplos de conversaciones reales o simuladas, como consultas de financiamiento o recomendaciones de autos. Tras cada cambio, estas pruebas comparan las respuestas actuales con las esperadas y detectan cualquier diferencia.

# Roadmap y Backlog

## Customization

Al proyecto se le deben agregar configuraciones de sistema como poder configurar que modelos se usan en los prompts. Poder configurar las temperaturas. Y tambien poder editar los prompts sin tener que modificar el codigo.

## Seguridad

Tenemos que crear un WebHook Authentication class en Django para prevenir ataques externos. Podemos pasar un token al WH en Twilio para validar que el request venga un sender validado.

### Kubernetes

Se debe implementar un setup de Kubernetes para poder crear mas pods de Django o Celery en caso de que la app tenga demasiada demanda. De esta manera se garantiza la estabilidad y flexibiliad del producto en el entorno de producción.

## Vector Store

Para optimizar la eficiencia, se debe crear un Vector Store (Elasticsearch, Pinecone, etc.) y poblarlo con las características de los vehículos disponibles y los artículos de conocimiento de Kavak. Para ello, podemos utilizar los embeddings de OpenAI. Cada caracterisitca de vehículo o fragmento de artículo se convertirá en un vector mediante el modelo de embeddings de OpenAI y se indexará en el Vector Store.

Cuando un usuario realice una consulta, primero consultaremos este almacén vectorial para recuperar los documentos más relevantes y, así, proporcionarle al LLM un contexto preciso y enriquecido antes de invocar la generación de la respuesta.

## MVP Interno

En esta fase se desplegará el pipeline básico. Normalización de entrada, RAG de vehículos y prompt final. El objetivo es validar el flujo mínimo de recomendaciones.

## Validación Interna de QA

Se implementarán mecanismos de manejo de errores, y se desarrollarán pruebas unitarias e automatizadas para asegurar la calidad del código y la fiabilidad del bot.

## Pruebas de Usuario

Un grupo reducido de clientes utilizará el agente. Se capturará su feedback sobre relevancia, tiempos de respuesta y usabilidad. Con base en estos comentarios, se ajustarán prompts y parámetros del LLM.

## Lanzamiento Público

Se realizará el despliegue en producción usando AWS. El proceso de CI/CD quedará automatizado con GitHub Actions. Se configurarán monitorización y alertas para garantizar estabilidad y visibilidad de métricas en tiempo real usando herramientas de AWS.

# Manual de Instalación

Prerrequisitos

- Git (para clonar el repositorio)
- Docker & Docker Compose (v1.27+)
- ngrok (para exponer webhooks locales)
- Clave API de OpenAI

## 1. Clonar el Repositorio

git clone https://github.com/edgarlias/agentic_chatbot.git
cd agentic_chatbot

## 2. Variables de Entorno

Crea un archivo .env en la raíz del proyecto con el siguiente contenido:

```
# Django
DJANGO_SECRET_KEY=your_django_secret_key
DJANGO_DEBUG=True
DJANGO_ALLOWED_HOSTS=localhost,127.0.0.1

# Base de datos (Postgres)
POSTGRES_DB=agentdb
POSTGRES_USER=agentuser
POSTGRES_PASSWORD=secret

# OpenAI
OPENAI_API_KEY=sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
# (Opcional) Redis
REDIS_URL=redis://redis:6379/0
```

### 3. Construir y Ejecutar con Docker Compose

```
# 1. Construir imágenes e iniciar servicios en segundo plano
docker-compose up -d --build

# 2. Crear tablas de base de datos
docker-compose exec web python manage.py migrate

# 3. (Opcional) Recopilar archivos estáticos
docker-compose exec web python manage.py collectstatic --noinput

# 4. Crear superusuario
docker-compose exec web python manage.py createsuperuser

# 5. Iniciar el proyecto y ver logs

docker compose up -d && docker compose logs -f
```

### 4. Exponer Localmente con ngrok

```
# Iniciar ngrok en el puerto 8000
ngrok http 8000
```

Actualiza tu webhook de prueba (por ejemplo en Twilio) con la URL HTTPS de ngrok: https://<tunnel>.ngrok.io/api/chat/twilio/inbound/

# Deployment

## Infraestructura en AWS

- Repositorio de imágenes: Amazon ECR para almacenar las imágenes Docker.
- Base de datos: Amazon RDS (PostgreSQL)

## Kubernetes

- Crear clúster gestionado con Terraform.

## Gestión de secretos (AWS Secrets Manager)

- Almacenar claves sensibles (DJANGO_SECRET_KEY, credenciales de RDS, APIs externas, etc.) en Secrets Manager.

## Monitoreo y registro (CloudWatch)

- Enviar logs de aplicación y contenedores a CloudWatch Logs.
- Establecer alarmas básicas (uso CPU, memoria, errores HTTP).

## Mejoras propuestas

- Pipelines de CI/CD automatizados (por ejemplo, GitHub Actions):
  - Pruebas y lint al crear PR.
  - Build y push de imagen a ECR.
- Separar configuraciones por entorno (dev/qa/prod/) en Django.
- Implementar autoscaling de pods de Django o Celery Workers según carga.

Con esta arquitectura, se garantiza un despliegue seguro, escalable y fácil de mantener, aprovechando los servicios gestionados de AWS.