MEMORIA DEL PROYECTO

Asistente Turístico de Córdoba con IA Conversacional (RAG + Voz)

1. Introducción

En el contexto actual, los sistemas de **inteligencia artificial conversacional** se han convertido en una herramienta clave para mejorar la interacción entre usuarios y servicios digitales. En especial, los asistentes virtuales aplicados al ámbito turístico permiten ofrecer información personalizada, accesible y disponible en cualquier momento, mejorando la experiencia del visitante.

El presente proyecto consiste en el desarrollo de un **asistente turístico inteligente de la ciudad de Córdoba**, capaz de responder preguntas tanto por **texto como por voz**, utilizando técnicas modernas de **Recuperación Aumentada por Generación (RAG)** y servicios de **procesamiento de lenguaje natural y voz**.

El asistente permite al usuario realizar consultas sobre monumentos, rutas, eventos, gastronomía y otros aspectos turísticos de Córdoba, obteniendo respuestas basadas **exclusivamente en información previamente cargada y validada**, evitando así respuestas inventadas o no verificadas.

Este proyecto se ha desarrollado como **evaluación final de la asignatura Herramientas de IA Clásica**, integrando múltiples tecnologías vistas a lo largo tanto del módulo como del máster en sí, cumpliendo los requisitos de una solución **end-to-end**, desde la ingesta de datos hasta la interacción conversacional por voz.

---

2. Objetivos del proyecto

2.1 Objetivo general

Desarrollar un **asistente conversacional turístico inteligente**, accesible por texto y voz, que proporcione información fiable sobre la ciudad de Córdoba mediante técnicas de **RAG (Retrieval-Augmented Generation)** y servicios de **Speech-to-Text (STT)** y **Text-to-Speech (TTS)**.

2.2 Objetivos específicos

- Implementar una **API backend** basada en FastAPI para gestionar las consultas del asistente.
- Construir una **base de conocimiento vectorial** a partir de documentos turísticos reales.
- Integrar un **modelo de lenguaje (Gemini)** para generar respuestas naturales basadas en contexto recuperado.
- Incorporar **reconocimiento de voz (STT)** y **síntesis de voz (TTS)** mediante Azure Cognitive Services.
- Conectar el asistente con **Dialogflow ES** como agente conversacional.
- Integrar el sistema con **Telegram**, permitiendo interacción por texto y notas de voz.
- Garantizar respuestas controladas, evitando alucinaciones del modelo.
- Documentar completamente el proyecto y su arquitectura.

---

3. Alcance del proyecto

El proyecto cubre el siguiente **alcance funcional**:

- **Consultas turísticas en lenguaje natural.**
- **Respuestas basadas únicamente en información previamente indexada.**
- **Interacción por:**
    - **Texto** (API, Dialogflow, Telegram).
    - **Voz** (Telegram y endpoint de voz).
- **Persistencia de información** mediante una **base de datos vectorial local.**
- **Ejecución en entorno local** mediante **Docker y Python**.

Este alcance define un **asistente turístico centrado en la consulta informativa**, sin funcionalidades transaccionales ni acceso a fuentes externas en tiempo real.

---

4. Arquitectura general del sistema

El sistema sigue una **arquitectura modular y desacoplada**, basada en servicios, que permite separar claramente cada responsabilidad.

Componentes principales:

1. **Usuario final**
    - Interactúa mediante texto o voz.
    - Canales: Dialogflow, Telegram o API directa.
2. **Agente conversacional (Dialogflow ES)**
    - Detecta intenciones del usuario.
    - Redirige las consultas dinámicas al webhook.
3. **API Backend (FastAPI)**
    - Orquesta todo el flujo y centraliza la lógica del sistema. Incluye documentación Swagger en /docs.
    - Expone endpoints REST (/ask, /voice, /fulfillment, etc.).
4. **Pipeline RAG**
    - Recuperación de contexto en Qdrant.
    - Generación de respuestas con Gemini.
5. **Base vectorial (Qdrant)**
    - Almacena embeddings de documentos turísticos.
    - Permite búsquedas semánticas eficientes.
6. **Servicios de Voz (Azure Cognitive Services)**
    - STT: audio → texto.
    - TTS: texto → audio.
7. **Telegram Bot**
    - Canal de entrada y salida para texto y audio.

Esta separación de componentes facilita el mantenimiento, la evolución del sistema y la posibilidad de sustituir tecnologías concretas sin afectar al resto de la arquitectura.

4.1. Diagrama del flujo del sistema

A continuación se muestra el **flujo completo de funcionamiento del asistente turístico**, desde la **interacción del usuario** hasta la **generación de la respuesta final**.

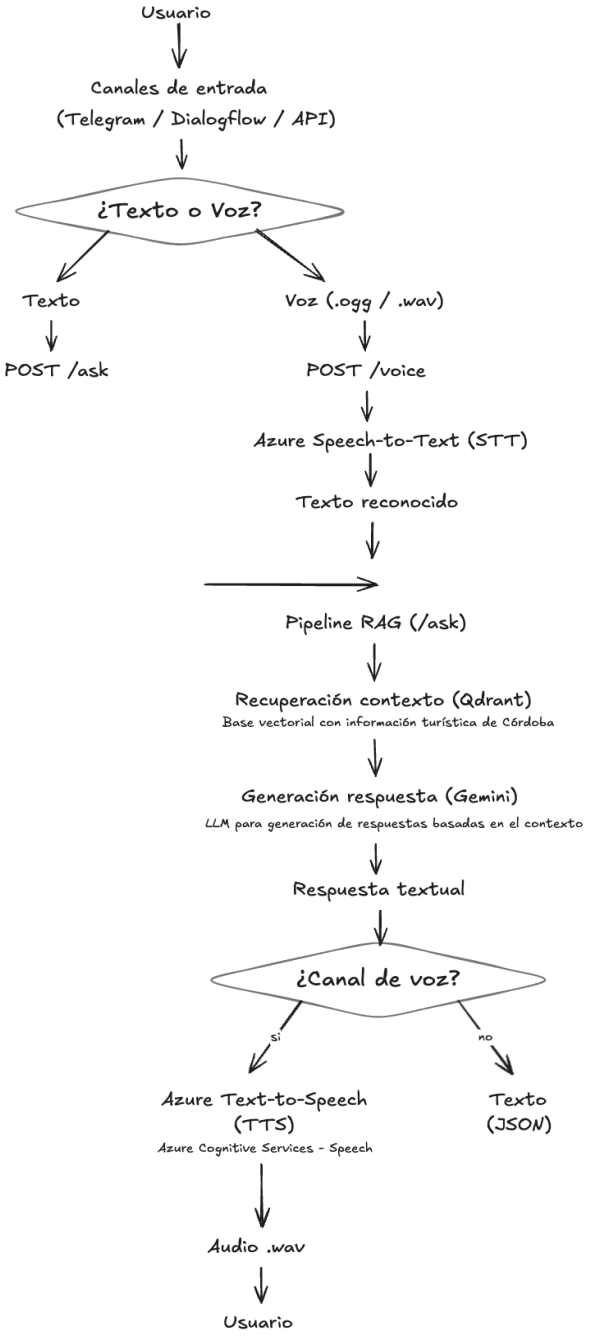
El usuario puede interactuar con el sistema mediante **texto o voz** a través de diferentes **canales**, como **Telegram**, **Dialogflow** o una **API REST directa**.

En el caso de **interacción por voz**, el audio se envía al endpoint **/voice**, donde se utiliza **Azure Cognitive Services (Speech-to-Text)** para convertir la voz en texto. El texto resultante se envía al endpoint **/ask**, donde se ejecuta el **pipeline RAG**.

El **pipeline RAG** realiza una **recuperación de contexto relevante** desde la **base de datos vectorial Qdrant**, que contiene información turística de la ciudad de Córdoba, y posteriormente genera una **respuesta contextualizada** utilizando el **modelo de lenguaje Gemini**.

Si la interacción original fue por **voz**, la respuesta textual se convierte nuevamente en **audio** mediante **Azure Text-to-Speech** y se devuelve al usuario en formato **.wav**. En caso contrario, la respuesta se devuelve directamente en **formato texto**. Este flujo permite **unificar la lógica de procesamiento** independientemente del canal de entrada.

A continuación se muestra el **diagrama del flujo del sistema**, que representa visualmente el proceso descrito:



4.2. Arquitectura del sistema y componentes

El sistema se ha implementado como una **API backend en FastAPI** que centraliza la lógica del asistente turístico y expone endpoints para consultas por texto y por voz. La solución integra **servicios externos** (Gemini y Azure Speech) y una **base de datos vectorial** (Qdrant) para realizar recuperación de contexto (RAG).

En la siguiente imagen se muestra la **documentación Swagger** generada automáticamente por **FastAPI**, donde se pueden observar los **endpoints** expuestos por el sistema.

Componentes principales:

1) API Backend (FastAPI)

- Actúa como punto central del sistema y expone los endpoints principales:
    - **GET /health:** verificación de estado (**Gemini** configurado y **Qdrant** accesible).
    - **POST /ask:** endpoint principal del **pipeline RAG** (pregunta → recuperación → respuesta).
    - **POST /voice:** endpoint de **interacción por voz** (audio .wav → STT → RAG → TTS → audio .wav).
    - **POST /fulfillment: webhook para Dialogflow** (texto → RAG → respuesta).
- Incluye configuración de CORS y documentación automática en Swagger (/docs).

2) Canales de entrada / Integraciones

- **Telegram**: integración mediante webhook (/telegram/webhook), soporta:
    - Mensajes de texto: se redirigen al flujo de RAG (/ask).

    - Mensajes de voz (nota de voz/audio): se descargan desde Telegram, se convierten a WAV 16 kHz mono y se procesan como voz.

- **Dialogflow**: canal de texto a través del endpoint /fulfillment, que recibe el mensaje del usuario desde el intent y devuelve fulfillmentText.

- **API REST directa**: el usuario puede consumir directamente /ask o /voice desde Swagger o herramientas como curl/Postman.

3) Pipeline RAG (Recuperación + Generación)

- La consulta del usuario se procesa en POST /ask:

    1. **Embeddings** de la pregunta usando el modelo intfloat/multilingual-e5-small.

    2. **Recuperación de contexto** en **Qdrant** (búsqueda vectorial top-k).

    3. Construcción de un **prompt** con el contexto recuperado.

    4. **Generación de respuesta** usando **Gemini**, restringiendo la respuesta al contexto recuperado.

- El endpoint devuelve respuesta y un listado de fuentes (sources) para trazabilidad.

4) Base de datos vectorial (Qdrant)

- Qdrant almacena los embeddings del contenido turístico (chunks) junto con metadatos como source y chunk_id.

- Se ejecuta en local mediante Docker Compose, y mantiene persistencia de datos en el directorio qdrant_data/.

- El contenido se prepara mediante scripts de ingestión y chunking, generando un fichero chunks.jsonl que posteriormente se inserta en Qdrant.

5) Servicios de voz (Azure Cognitive Services Speech)

- **Speech-to-Text (STT)**: convierte audio WAV (16 kHz mono) en texto.

- **Text-to-Speech (TTS)**: convierte la respuesta generada en audio WAV (16 kHz mono).

- Se usa en el endpoint /voice y también en el canal Telegram cuando el usuario envía audio.

Flujo general de comunicaciones

- FastAPI coordina el flujo completo: recibe la entrada (texto/voz), llama a Qdrant para recuperar contexto, llama a Gemini para generar respuesta y, si el canal es de voz, llama a Azure Speech para generar audio final.

- El sistema está diseñado para que **el núcleo sea siempre el mismo (/ask)**, independientemente del canal de entrada, lo que facilita mantenimiento y extensibilidad.

4.3. Tecnologías utilizadas

Para el desarrollo del asistente turístico se han utilizado distintas tecnologías y servicios, combinando herramientas de backend, bases de datos vectoriales y servicios de inteligencia artificial en la nube.

Backend y API

- **FastAPI**: framework principal para la creación de la API REST. Permite definir endpoints, validación de datos, gestión de peticiones asíncronas y documentación automática con Swagger.

- **Uvicorn**: servidor ASGI utilizado para ejecutar la aplicación en local.

- **Python 3**: lenguaje principal de desarrollo.

Recuperación de información (RAG)

- **Qdrant**: base de datos vectorial utilizada para almacenar embeddings de información turística de Córdoba y realizar búsquedas semánticas.

- **Sentence-Transformers** (intfloat/multilingual-e5-small): modelo de embeddings multilingüe optimizado para español, utilizado para representar textos como vectores.

- **Pipeline RAG**: combinación de recuperación de contexto (Qdrant) y generación de respuestas basadas únicamente en la información recuperada.

Modelo de lenguaje

- **Gemini (Google Generative AI)**: modelo de lenguaje utilizado para generar respuestas en lenguaje natural a partir del contexto recuperado, evitando respuestas fuera del dominio turístico cargado.

Procesamiento de voz

- **Azure Cognitive Services – Speech**:
    - **Speech-to-Text (STT)**: conversión de audio WAV (16 kHz, mono) a texto.
    - **Text-to-Speech (TTS)**: generación de audio WAV a partir de la respuesta textual.
- **ffmpeg**: herramienta utilizada para la conversión de audios (OGG → WAV) en el canal de Telegram.

Integraciones externas

- **Telegram Bot API**: canal de interacción con el usuario, soportando mensajes de texto y notas de voz.
- **Dialogflow**: plataforma de NLU utilizada como canal de texto adicional mediante webhook.
- **API REST directa**: consumo del sistema mediante endpoints expuestos por FastAPI.

Infraestructura y herramientas auxiliares

- **Docker & Docker Compose**: ejecución de Qdrant en contenedor con persistencia de datos.
- **Makefile**: automatización de tareas habituales (creación de entorno, ingestión de datos, arranque de la API).
- **Swagger / OpenAPI**: documentación interactiva de la API (/docs).
- **Git**: control de versiones del proyecto.

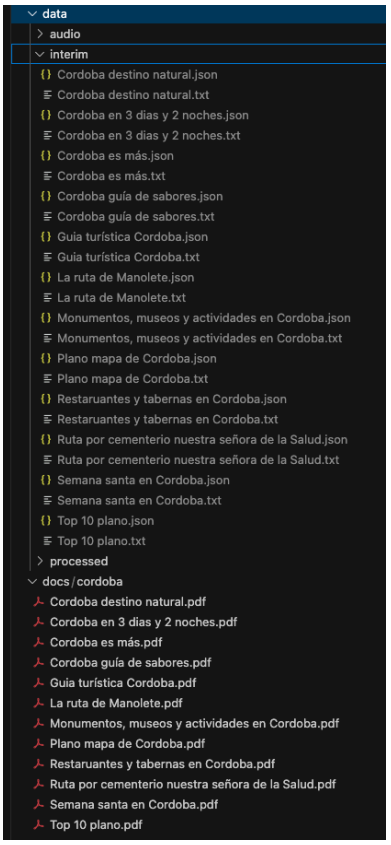
---

5. Desarrollo e implementación

En este apartado se describe el proceso de desarrollo del asistente turístico, detallando cómo se ha construido el sistema desde la ingestión de datos hasta la generación de respuestas, incluyendo la integración de voz y los distintos canales de entrada.

5.1 Preparación e ingestión de datos

El sistema se basa en información turística de la ciudad de Córdoba obtenida a partir de documentos en formato PDF (guías turísticas, rutas, monumentos, gastronomía, etc.).



El proceso de ingestión se ha dividido en varias fases:

1. **Extracción de texto**
    - Los PDFs originales se almacenan en la carpeta docs/cordoba.
    - Mediante el script ingest_pdf.py, el contenido textual se extrae y se guarda en la carpeta data/interim tanto en formato .txt como .json.
2. **Normalización y limpieza**
    - Se eliminan caracteres innecesarios, saltos de línea redundantes y se normaliza el texto para facilitar su posterior procesamiento.
    - Esta fase garantiza una mejor calidad de los embeddings y de la recuperación semántica.

Este enfoque permite desacoplar los datos originales del procesamiento posterior y facilita la reutilización de los textos.

5.2 Segmentación del texto (Chunking)

Debido a las limitaciones de tamaño de contexto de los modelos de lenguaje y a las buenas prácticas en sistemas RAG, los textos largos se dividen en fragmentos más pequeños o *chunks*.

- El proceso se realiza mediante el módulo chunking.py.
- Cada documento se divide en fragmentos de tamaño controlado, con un pequeño solapamiento entre ellos para no perder contexto.
- El resultado final se almacena en data/processed/chunks.jsonl.

Este paso es clave para:

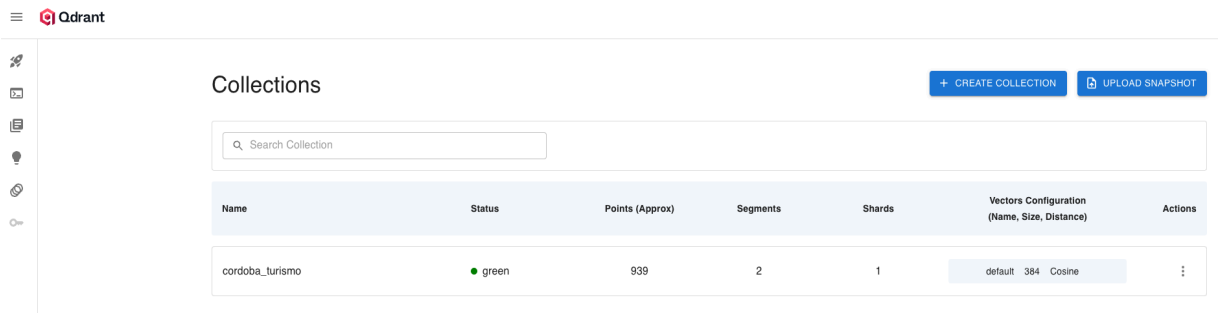
- Mejorar la precisión de las búsquedas en la base vectorial.
- Evitar respuestas incompletas o fuera de contexto.

5.3 Generación de embeddings y almacenamiento en Qdrant

Una vez generados los fragmentos de texto, se procede a su vectorización:

- Se utiliza el modelo **intfloat/multilingual-e5-small**, optimizado para textos en español.
- Cada fragmento se convierte en un vector numérico (embedding).
- Los embeddings, junto con metadatos como la fuente y el identificador del fragmento, se almacenan en **Qdrant**, la base de datos vectorial.

La colección se crea una única vez y se reutiliza durante toda la ejecución del sistema, permitiendo búsquedas semánticas eficientes basadas en similitud.



5.4 Implementación del pipeline RAG

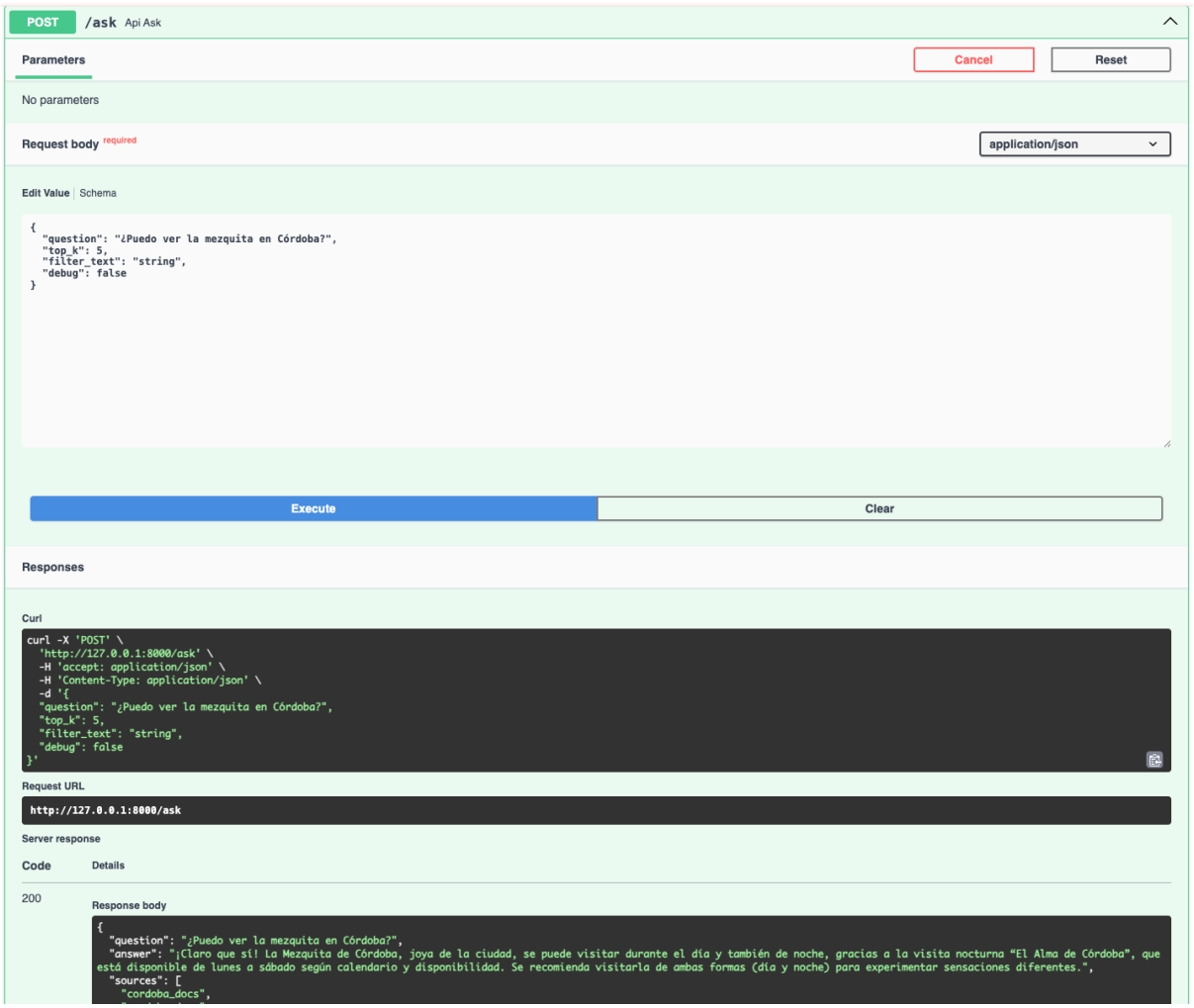
El núcleo del sistema es el endpoint /ask, que implementa el pipeline **Retrieval-Augmented Generation (RAG)**:

1. **Recepción de la pregunta del usuario**.
2. **Cálculo del embedding de la pregunta**.
3. **Recuperación de contexto relevante** desde Qdrant mediante búsqueda semántica.
4. **Construcción del prompt** con:
    - La pregunta original.

- El contexto recuperado.

5. **Generación de la respuesta** utilizando el modelo **Gemini**, restringiendo la respuesta únicamente a la información proporcionada en el contexto.

Este enfoque evita respuestas inventadas y asegura que el asistente actúe como un guía turístico basado en información real.



5.5 Integración de voz (STT y TTS)

El sistema soporta interacción por voz gracias a **Azure Cognitive Services – Speech**:

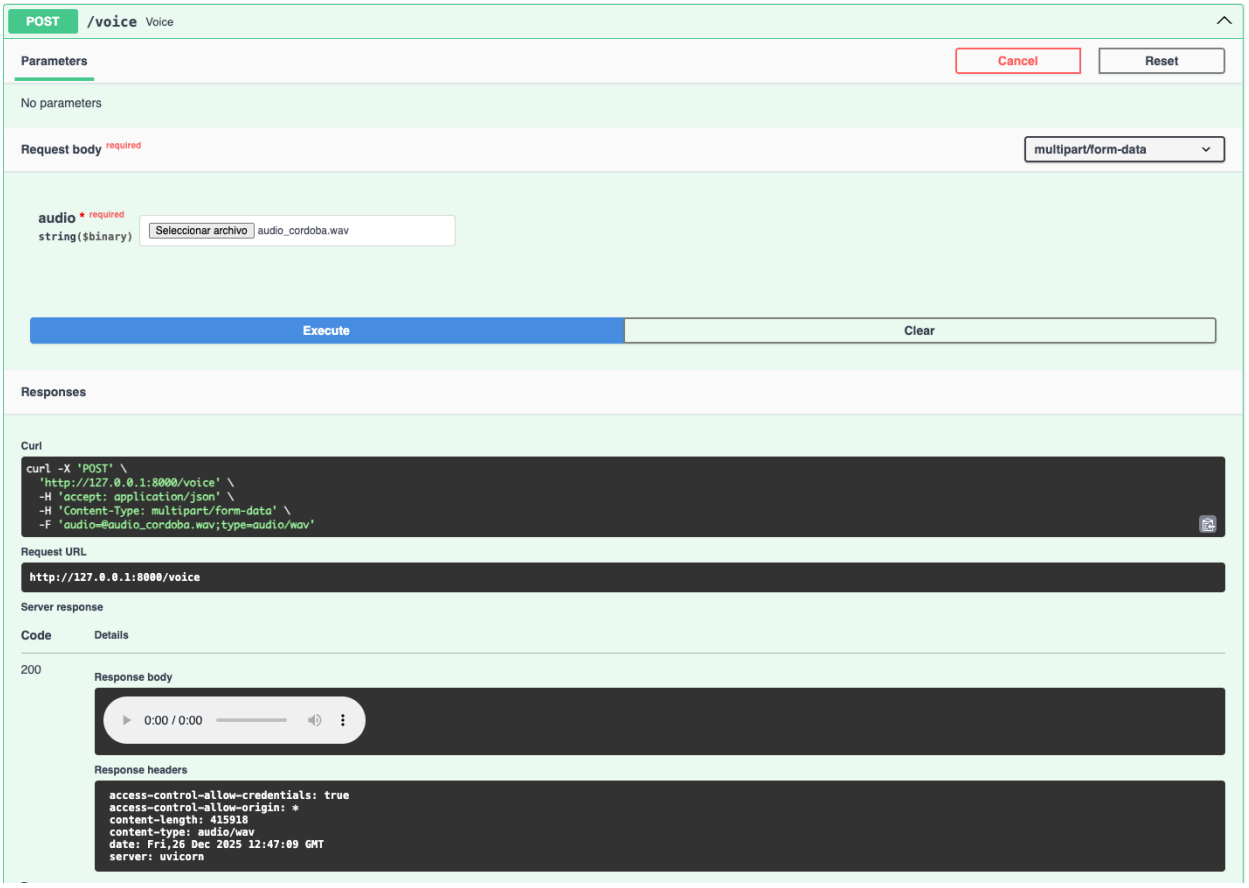
- **Speech-to-Text (STT)**:
    - El endpoint /voice recibe un archivo WAV (16 kHz, mono).
    - Azure convierte el audio a texto en español.

    - El texto resultante se envía al pipeline RAG.

- **Text-to-Speech (TTS)**:

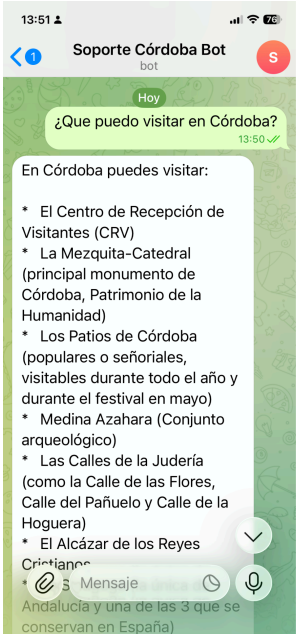
    - La respuesta textual generada se transforma nuevamente en audio WAV.
    - Se utiliza una voz neuronal en español.
    - El audio final se devuelve al usuario.

Este flujo permite una experiencia conversacional completa por voz.



La imagen anterior muestra la ejecución del endpoint /voice desde la documentación Swagger. Al enviar un archivo WAV, el sistema ejecuta el flujo completo de Speech-to-Text, pipeline RAG y Text-to-Speech, devolviendo un archivo de audio WAV con la respuesta generada. Aunque Swagger no reproduce directamente el audio devuelto, el encabezado content-type: audio/wav y el código de respuesta 200 confirman la correcta generación del audio.

5.6 Integración con canales externos



El asistente se ha integrado con distintos canales de entrada:

- **Telegram**:
  - Mediante un bot que soporta mensajes de texto y notas de voz.
  - En el caso de audio, se realiza una conversión previa de OGG a WAV.

- **Dialogflow**:
  - Integración mediante webhook (/fulfillment).
  - Permite usar el asistente como backend de un agente conversacional.

- **API REST directa**:
  - Acceso a través de Swagger (/docs) o peticiones HTTP.

Esta arquitectura multicanal permite reutilizar el mismo backend para distintos tipos de interfaz.

5.7 Automatización y ejecución

Para facilitar el uso del proyecto se ha incluido un **Makefile**, que permite:

- Crear el entorno virtual.
- Instalar dependencias.
- Ejecutar los scripts de ingestión y carga de datos.
- Lanzar la API en local.

Además, Qdrant se ejecuta mediante **Docker Compose**, garantizando persistencia de datos y facilidad de despliegue.

---

6. Conclusiones

Este proyecto demuestra la viabilidad de construir un **asistente turístico inteligente y multimodal**, integrando tecnologías modernas de IA generativa, bases vectoriales y servicios de voz.

La solución desarrollada cumple con los objetivos planteados, ofreciendo respuestas fiables, controladas y accesibles tanto por texto como por voz, y constituye una base sólida para futuras ampliaciones, como despliegue en la nube o personalización por usuario.