LangChain

https://github.com/MartinCastroAlvarez/langchain-virtual-assistant

Martín Alejandro Castro Álvarez

martincastro.10.5@gmail.com

Universidad Internacional de Valencia (VIU)

Calle Pintor sorolla, 21 46002, Valencia (España)

Mayo 2025

Abstract. Este proyecto presenta un asistente virtual médico construido con LangChain que analiza PDFs de consultas clínicas, identifica condiciones similares y ofrece recomendaciones en tiempo real. El sistema fue diseñado para abordar los problemas del Centro Médico La Nueva Esperanza, incluyendo la sobrecarga cognitiva del personal, los largos tiempos de consulta y la falta de acceso ágil a la información médica. Utiliza embeddings semánticos, recuperación de información mediante RAG, procesamiento multilingüe con GPT-3.5-turbo y herramientas personalizadas como traducción y división de síntomas. El agente es capaz de interpretar preguntas, recuperar información contextual y entregar respuestas precisas y adaptadas al usuario. Las pruebas demostraron alta precisión, buena experiencia de usuario y tiempos de respuesta óptimos, validando su viabilidad dentro del presupuesto y plazo definidos.

Keywords: LangChain, LLM, GPT, OpenAI, Python, Information Retrieval, Smart Agent.

1. Introducción

1.1. Objetivo

Este informe presenta una solución basada en LangChain [3] orientada al desarrollo de un asistente virtual médico, capaz de analizar documentos PDF de consultas médicas, identificar condiciones similares y sugerir recomendaciones diagnósticas o terapéuticas.

1.2 Necesidad de Negocio

Esta solución (Ver Fig. 11) responde a los principales desafíos del Centro Médico La Nueva Esperanza, que atiende a más de 35.000 pacientes y enfrenta sobrecarga cognitiva, consultas extensas y dificultad para acceder rápidamente a la información. El sistema busca facilitar el acceso a datos médicos relevantes, estructurar historiales clínicos, y ofrecer recomendaciones basadas en casos similares, todo con respuestas claras en distintos idiomas, optimizando recursos dentro de un presupuesto de 75.000 euros.

2. Problema

2.1. Descripción detallada

El sistema debe poder procesar y analizar PDFs médicos que no siguen una estructura estándarizada, identificar síntomas y condiciones presentes en los textos, buscar similitudes con casos anteriores sin revelar datos sensibles, y generar recomendaciones en base a tratamientos que hayan funcionado en el pasado. Además, tiene que adaptar el lenguaje de las respuestas al nivel de comprensión del usuario, integrarse sin fricciones con los sistemas actuales de Historia Clínica Electrónica, cumplir con normativas de protección de datos como el RGPD, y todo esto con un presupuesto limitado y un plazo de ejecución de solo cuatro meses.

2.2. Público objetivo

El sistema está pensado para pacientes que buscan entender mejor su situación médica, médicos que necesitan comparar casos similares y reducir su carga administrativa, y también administradores que quieren mejorar la eficiencia operativa de la clínica. Además, el foco principal está en el Centro Médico La Nueva Esperanza, pero el enfoque es aplicable a otras instituciones con necesidades similares.

2.3. Soluciones existentes

Actualmente existen herramientas como gestores clásicos de historiales médicos, asistentes virtuales médicos genéricos o buscadores de papers médicos. Sin embargo, la mayoría no están integrados con LLMs, no manejan bien consultas en lenguaje natural, ni están optimizados para ofrecer recomendaciones personalizadas desde historiales. La mayoría utilizan Natural Language Processing (NLP) [1] tradicional, que tiene resultados muy limitados en la práctica. Aparte, tampoco tienen buenos mecanismos de indexación de bases de datos médicas o farmacológicas, lo cual limita mucho su utilidad en tiempo real.

3. Solución

3.1. Arquitectura

3.1.1 generator.py

El archivo *generator.py* (Fig. 1) se encarga de crear automáticamente PDFs que simulan consultas médicas reales, utilizando una estructura estandarizada y datos médicos predefinidos. Estos documentos sirven como base para el sistema, permitiendo entrenar y validar las capacidades de análisis y recuperación de información. Además, su diseño contempla la posibilidad de integrarse con sistemas reales de Historia Clínica Electrónica. Los PDFs generados incluyen metadatos médicos relevantes, están preparados para soportar múltiples idiomas y se validan internamente para asegurar coherencia en los datos.

3.1.2 indexer.py

El archivo *indexer.py* (Fig. 2) toma los PDFs generados, limpia y normaliza el texto médico, eliminando partes irrelevantes como encabezados repetitivos y redundantes, y luego los divide en segmentos

optimizados para mantener el contexto clínico. A cada *chunk* se le genera su embedding semántico usando un modelo pre-entrenado, y toda la información se guarda en un *vectorstore.json* (Fig. 3.) que permite realizar búsquedas por similitud o función de distancia. Esto permite que el sistema pueda identificar con rapidez y precisión casos similares en el historial médico sin comprometer datos sensibles.

3.1.3 agent.py

El archivo *agent.py* (Fig. 8) es el agente que se comunica directamente con el usuario, interpretando sus preguntas y coordinando las acciones necesarias para responder de forma útil y personalizada. Utiliza un sistema de memoria para mantener el contexto a lo largo de la conversación, maneja múltiples condiciones médicas a la vez y tiene acceso a herramientas especializadas como traductores, buscadores web y divisores de síntomas. Además, las respuestas se adaptan automáticamente al idioma del usuario, garantizando una experiencia de usuario eficiente [2].

3.2. Funcionamiento

3.2.1 Retrieval Augment Generation (RAG)

El sistema de embeddings utiliza el modelo *SentenceTransformer all-MiniLM-L6-v2* (Fig. 4), elegido por su equilibrio entre velocidad y precisión en tareas semánticas. Cada segmento de texto extraído de los PDFs se transforma en un vector de 384 dimensiones. Estos vectores se almacenan en *vectorstore.json* (Fig. 3) que actúa como base de datos vectorial, optimizado para consultas médicas. Se utiliza una caché en */tmp/sentence_transformers* (Fig. 4) para evitar recomputaciones innecesarias, y se normalizan todos los vectores para asegurar coherencia en los cálculos de similitud, que se basan en la distancia coseno.

3.2.2 Information Retrieval (IR)

Para recuperar la información más relevante, se emplea un retriever llamado *PDFVectorRetriever* (Fig. 7), que compara los embeddings de la consulta del usuario con los almacenados en el *vectorstore.json*. Esta búsqueda semántica permite filtrar por relevancia y aplicar un ranking personalizado. El sistema admite múltiples idiomas y está configurado para devolver hasta tres resultados por defecto (Fig. 5). Además, implementa mecanismos de filtrado por similitud y caché de resultados, lo que mejora tanto la precisión como la velocidad de respuesta del sistema.

3.2.3 Natural Language Processing (NLP)

Se utiliza el modelo *GPT-3.5-turbo* (Fig. 5), con prompts diseñados específicamente para contextos médicos. Este modelo se encarga de interpretar las preguntas del usuario, generar respuestas informadas y traducirlas si es necesario, adaptándose al idioma en que fue realizada la consulta. El sistema puede manejar múltiples condiciones médicas dentro de una misma interacción, manteniendo el contexto mediante *ConversationSummaryBufferMemory* (Fig. 6).

4. LangChain

4.1 Carga de Documentos

Para la carga y preparación de los PDFs médicos, se utiliza *PyPDFLoader* (Fig. 5), que permite extraer el contenido textual de manera confiable. Una vez cargado, el texto se procesa mediante un splitter recursivo que divide el contenido en chunks de 500 caracteres con *sliding window*, lo que garantiza que no se pierda contexto relevante entre segmentos.

4.2 Embeddings

La generación de embeddings se realiza con *SentenceTransformer* (Fig. 4). Cada chunk del documento se transforma en un vector que se guarda en *vectorstore.json* (Fig. 3). Para evitar latencias innecesarias en ejecuciones repetidas, se define una caché en el directorio */tmp/sentence_transformers*, lo que permite acelerar el pipeline de indexación y recuperación.

4.3 PromptTemplate

Uno de los prompts se encarga de las tareas RAG [3], otro Prompt traduce textos médicos del inglés al español (Fig. 9), y un tercer Prompt descompone síntomas complejos en problemas individuales (Fig. 5). Estos templates permiten guiar al modelo de lenguaje con instrucciones específicas para cada contexto, y están estructurados usando *PromptTemplate* de *LangChain* [3], lo que facilita su reutilización y ajuste.

4.4 Information Retrieval

El sistema de recuperación está compuesto por un *PDFVectorRetriever* (Fig 8) personalizado, que permite realizar búsquedas semánticas sobre los embeddings de los documentos. Implementa técnicas de ranking por similitud coseno, filtrado de duplicados y normalización de puntuaciones para mejorar la relevancia de los resultados.

4.5 Agente

El agente (Fig. 8) integra múltiples herramientas y gestiona la interacción con el usuario final. Está configurado con una memoria conversacional para mantener el contexto de la conversación, utiliza *initialize_agent* de *LangChain*. Su diseño modular permite integrar nuevas tools (Fig. 9), manteniendo la lógica central enfocada en entregar respuestas útiles, precisas y adaptadas al lenguaje del usuario.

4.6 Tools

El agente implementa un conjunto de tools (Fig. 9) que extienden sus capacidades. La herramienta *SearchWeb* permite realizar búsquedas en tiempo real en la web para complementar la información de los PDFs, mientras que *TranslateToSpanish* se encarga de adaptar el lenguaje médico al español de manera comprensible. Por otra parte, la herramienta *SplitProblems* descompone mensajes en partes, facilitando un análisis más preciso.

4.7 Memory

El agente (Fig. 8) implementa un sistema de memoria conversacional basado en *ConversationSummaryBufferMemory* (Fig. 8), que mantiene un registro de las interacciones con el usuario. Se establece un límite de 1000 tokens para controlar la longitud de las conversaciones, y se utiliza un sistema de caché que permite persistencia y continuidad en los diálogos médicos. Además, las conversaciones se guardan en el disco en */tmp/conversation_cache.pkl*. La memoria implementa un mecanismo de resúmen que condensa las interacciones previas, sin consumir demasiados tokens (que tienen un impacto directo en el costo del modelo).

5. Pruebas

5.1 Casos de prueba representativos

Para evaluar el funcionamiento del asistente virtual, se realizaron pruebas con distintos tipos de entradas que simulan consultas reales en un entorno clínico (Ver Fig. 10).

5.1.1 Caso de Prueba 1

En primer lugar, se probó una consulta directa y simple: *"Tengo dolor de cabeza"*. El sistema recuperó documentos con sintomatología coincidente, generó recomendaciones basadas en casos previos y ofreció una respuesta clara y relevante: *"Catalina Ríos fue la paciente con la queja de dolor de cabeza. Esta información se encuentra en el Informe de Consulta Médica de Catalina Ríos, con fecha 02/05/2024."*

5.1.2 Caso de Prueba 2

Se evaluó la consulta *"tengo dolor de hombro"*, y el modelo respondió: *"El paciente con dolor de hombro tiene un diagnóstico de Síndrome del intestino irritable (SII) y se le recomienda programar una ecografía para descartar complicaciones. Este diagnóstico y recomendación se encuentran en el informe de consulta médica de Daniel Ortega en el archivo 'Daniel_Ortega_07-06-2024.pdf'"*.

5.1.3 Caso de Prueba 3

Se testearon consultas con múltiples condiciones clínicas: *"Tengo ansiedad y depresión"*. El agente logró dividir correctamente el input en dos problemas distintos, realizó búsquedas independientes para cada uno y unificó los hallazgos en una respuesta completa, sin mezclar las recomendaciones ni perder precisión: *"El paciente con ansiedad y depresión puede encontrar información relevante en los archivos 'Sophia_Ortega_18-03-2025.pdf', 'Valentina_Ríos_14-09-2024.pdf', 'William_Ríos_20-01-2025.pdf', entre otros."*.

5.2 Resultados obtenidos

El sistema mostró un comportamiento robusto en los distintos escenarios de prueba. Fue capaz de identificar casos clínicos similares con alta precisión, generar respuestas relevantes con recomendaciones basadas en evidencia previa y mantener la coherencia en interacciones complejas con múltiples síntomas.

Además, se observó una reducción notable en los tiempos de respuesta y una mejora general en la eficiencia de consulta. Por lo tanto, el agente contribuye a aliviar la carga cognitiva del personal médico y optimizar tareas administrativas, especialmente en contextos de alta demanda como el del Centro Médico La Nueva Esperanza.

5.3 Limitaciones detectadas

Entre las limitaciones más relevantes, se identificó la fuerte dependencia del sistema respecto a la calidad de los PDFs procesados. Si el contenido está mal estructurado o contiene errores, el análisis puede verse afectado. Además, aunque el modelo de embeddings ofrece buen rendimiento general, no siempre captura matices clínicos complejos, lo que puede impactar en la calidad de las recomendaciones.

6. Conclusiones

6.1 Valoración del trabajo

El sistema desarrollado cumple con los objetivos propuestos: permite analizar PDFs médicos, identificar casos similares y ofrecer recomendaciones útiles en distintos idiomas. Mejora la eficiencia operativa, reduce la carga cognitiva de los médicos y funciona dentro del presupuesto y plazo definidos.

6.2 Propuestas de mejora

A nivel técnico, se podrían usar modelos de embeddings más avanzados y mejorar el ranking de documentos. Además, se sugiere integrar más fuentes médicas, añadir validación automática y desarrollar un sistema de feedback para garantizar la convergencia de los resultados. En términos de usabilidad, sería útil crear una interfaz web, visualizaciones y una app móvil para profesionales [3].

7. Referencias

[1] Rogman, D. (2021). Transformers for Natural Language Processing.

[2] Casado, P. E. (2023). UX Design.

[3] Auffarth, B. (2023). Generative AI with Lang Chain.

8. Anexos

langchain-virtual-assistant / generator.py

Code Blame 342 lines (324 l…

```python
@dataclass
class Consultation:
    patient: str
    doctor: str
    medicine: str
    problem: str
    diagnosis: str
    recommendation: str
    date: datetime

    @staticmethod
    def safe_text(text: str, width: int = 100) -> str:
        return "\n".join(textwrap.wrap(text, width))

    @staticmethod
    def generate() -> Consultation:
        return Consultation(
            patient=random.choice(PATIENTS),
            doctor=random.choice(DOCTORS),
            medicine=random.choice(MEDICINES) if random.random() < 0.5 else "",
            problem=random.choice(PROBLEMS),
            diagnosis=random.choice(DIAGNOSES) if DIAGNOSES else "Diagnóstico pendiente",
            recommendation=random.choice(RECOMMENDATIONS),
            date=datetime.now() - timedelta(days=random.randint(0, 365)),
        )

    def to_pdf(self, path: str = PDF_DIR) -> str:
        pdf = FPDF()
        pdf.set_margins(20, 20, 20)  # Left, Top, Right margins
        pdf.add_page()
        pdf.set_font("Arial", size=12)

        # Calculate effective width for multi_cell
        effective_width = pdf.w - pdf.l_margin - pdf.r_margin

        pdf.cell(effective_width, 10, txt="Informe de Consulta Médica", ln=True, align="C")
        pdf.ln(10)
        pdf.cell(effective_width, 10, txt=f"Fecha de consulta: {self.date.strftime('%d/%m/%Y')}", ln=True)
        pdf.cell(effective_width, 10, txt=f"Paciente: {self.patient}", ln=True)
        pdf.cell(effective_width, 10, txt=f"Médico: {self.doctor}", ln=True)
        pdf.ln(10)
        pdf.multi_cell(effective_width, 10, txt=self.safe_text(f"Motivo de la consulta: {self.problem}", width=60))
        pdf.multi_cell(effective_width, 10, txt=self.safe_text(f"Diagnóstico: {self.diagnosis}", width=60))
        pdf.multi_cell(effective_width, 10, txt=self.safe_text(f"Recomendación: {self.recommendation}", width=60))
        if self.medicine:
            pdf.multi_cell(effective_width, 10, txt=self.safe_text(f"Medicamento prescripto: {self.medicine}", width=60))
        filename = os.path.join(path, f"{self.patient.replace(' ', '_')}_{self.date.strftime('%d-%m-%Y')}.pdf")
        pdf.output(filename)
        print(f"PDF generado: {filename}")
        return filename
```

Fig. 1: Código de generator.py

langchain-virtual-assistant / indexer.py

Code | Blame | 150 lines (129 l…

```
 82  class Pdf:
 89      def clean(self, text: str) -> str:
106
107  @dataclass
108  class Indexer:
109      pdf_dir: str = field(default=PDF_DIR)
110      db_file: str = field(default=DATABASE_FILE)
111      model_name: str = field(default=EMBEDDING_MODEL)
112      cache_dir: str = field(default=CACHE_DIR)
113      _model: SentenceTransformer | None = None
114
115      @property
116      def model(self) -> SentenceTransformer:
117          if self._model is None:
118              print(f"Loading embedding model: {self.model_name}...")
119              self._model = SentenceTransformer(self.model_name, cache_folder=self.cache_dir)
120              print("Model loaded.")
121          return self._model
122
123      def run(self):
124          assert os.path.exists(self.pdf_dir), f"Error: Directory '{self.pdf_dir}' not found."
125          pdf_filepaths = glob.glob(os.path.join(self.pdf_dir, "*.pdf"))
126          assert pdf_filepaths, f"No PDF files found in '{self.pdf_dir}'."
127          documents: list[Document] = []
128          for i, filepath in enumerate(pdf_filepaths, 1):
129              print(f"Document {i}/{len(pdf_filepaths)}")
130              pdf = Pdf(filepath)
131              document = Document(filename=pdf.filename, text="", embeddings=[])
132              chunks = pdf.split()
133              if chunks:
134                  document.text = chunks[0]
135                  chunk_embeddings = self.model.encode(chunks, convert_to_numpy=True)
136                  document.embeddings = [embedding.tolist() for embedding in chunk_embeddings]
137                  documents.append(document)
138
139          print(f"Processed {len(documents)} documents")
140          print("Writing database...")
141          database = [doc.to_dict() for doc in documents]
142          with open(self.db_file, "w") as f:
143              json.dump(database, f, indent=4)
144
145          print(f"Database successfully created at {self.db_file}")
146
```

Fig. 2: Código de indexer.py

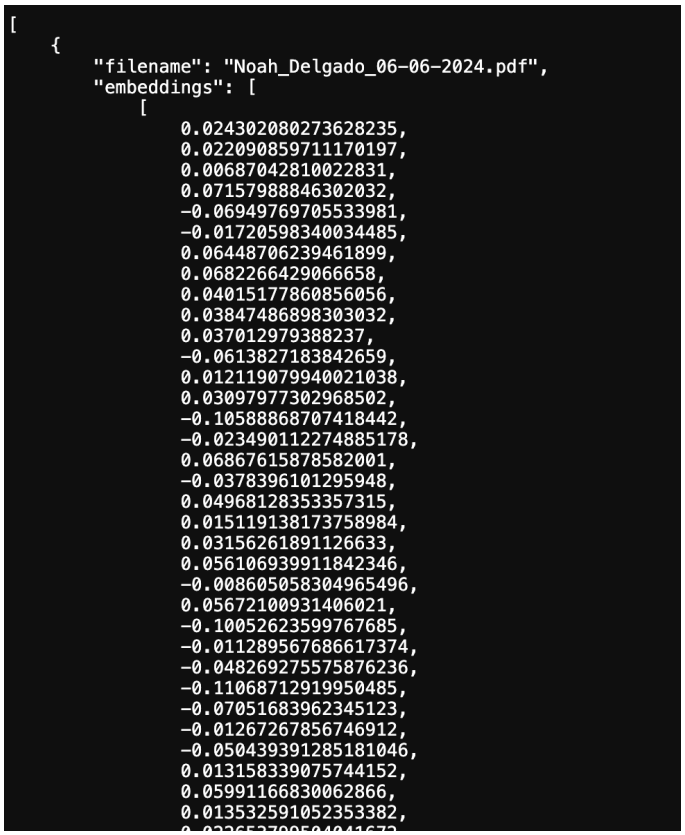

Fig. 3: Ejemplo de vectorstore.json

langchain-virtual-assistant / agent.py

Code Blame 406 lines (338 l…

```python

class Vector:
    model: SentenceTransformer | None = None

    @classmethod
    def load(cls) -> None:
        cls.model = SentenceTransformer(EMBEDDING_MODEL_NAME, cache_folder=CACHE_DIR)
        Out.green(f"Vector model loaded with {len(cls.model.encode('test'))} dimensions")

    @classmethod
    def distance(cls, query_embedding: np.ndarray, doc_embedding: np.ndarray) -> float:
        query_embedding = query_embedding.flatten()
        doc_embedding = doc_embedding.flatten()
        query_embedding = query_embedding.reshape(1, -1)
        doc_embedding = doc_embedding.reshape(1, -1)
        return float(cosine_similarity(query_embedding, doc_embedding)[0][0])

```

Fig. 4: Clase Vector, responsable de implementar cosine similarity.

langchain-virtual-assistant / agent.py

Code Blame 406 lines (338 l…

```python
class Store:
    @classmethod
    def load(cls, filepath: str = VECTORSTORE_FILE) -> None:
        assert os.path.exists(filepath), f"Vectorstore file not found at {filepath}. PDF RAG will not work."
        with open(filepath, "r") as f:
            raw_data = json.load(f)

        cls.db = []
        for item in raw_data:
            doc = Document.from_dict(item)
            if not doc.text:
                filepath = os.path.join(PDF_DIR, doc.filename)
                if os.path.exists(filepath):
                    loader = PyPDFLoader(filepath)
                    pages = loader.load()
                    doc.text = "\n".join(page.page_content for page in pages)
            cls.db.append(doc)

        total_embeddings = sum(len(doc.embeddings) for doc in cls.db)
        Out.green(f"Store loaded with {len(cls.db)} documents and {total_embeddings} total embeddings")

    @classmethod
    def search(cls, query_embedding: np.ndarray, n: int = 3) -> list[tuple[Document, float]]:
        assert cls.db, "Store.db has not been initialized."
        query_embedding = query_embedding.flatten()

        all_similarities: list[tuple[Document, float]] = []
        for doc in cls.db:
            for emb in doc.embeddings:
                similarity = Vector.distance(query_embedding, emb)
                all_similarities.append((doc, similarity))

        all_similarities.sort(key=lambda x: x[1], reverse=True)

        top_n_unique: list[tuple[Document, float]] = []
        seen_filenames = set()

        for doc, score in all_similarities:
            if doc.filename not in seen_filenames:
                seen_filenames.add(doc.filename)
                top_n_unique.append((doc, score))
                if len(top_n_unique) == n:
                    break

        Out.blue(f"Top {len(top_n_unique)} similarities: {[score for _, score in top_n_unique]}")
        Out.blue(f"Top {len(top_n_unique)} documents: {[doc.filename for doc, _ in top_n_unique]}")
        return top_n_unique
```

Fig. 5: Clase Store, responsable de búsqueda por RAG.

langchain-virtual-assistant / agent.py

Code Blame 406 lines (338 l…

```python

class Brain:
    model: ChatOpenAI | None = None

    @classmethod
    def load(cls) -> None:
        cls.model = ChatOpenAI(temperature=0, openai_api_key=OPENAI_API_KEY, model_name="gpt-3.5-turbo")
        Out.green(f"Brain model loaded with {cls.model.model_name}")


class Template:
    RAG = PromptTemplate(
        input_variables=["input", "context"],
        template=(
            "You are an assistant specializing in analyzing medical consultation PDFs. "
            "For each relevant medical case found in the PDFs, provide a comprehensive analysis including:\n"
            "1. The specific PDF documents that contain relevant information\n"
            "2. For each case found:\n"
            "   - The diagnosis given\n"
            "   - The treatment or medication prescribed\n"
            "   - The doctor's specific recommendations\n"
            "3. A summary comparing the cases if multiple relevant documents are found\n\n"
            "Important guidelines:\n"
            "- Always mention which PDF documents you're referencing\n"
            "- If multiple similar cases exist, compare their treatments and recommendations\n"
            "- If the information requested isn't found in the documents, clearly state this\n"
            "- Include any prescribed medications and their dosages\n"
            "- Emphasize that these are reference cases and the patient should consult a healthcare professional\n\n"
            "Context from PDF documents:\n{context}\n\n"
            "Question from the patient: {input}\n"
            "Your Analysis:"
        ),
    )
```

Fig. 5: Clase Brain, responsable de interactuar con la API de OpenAI.

langchain-virtual-assistant / agent.py

Code Blame 406 lines (338 l… Raw

```python

class Conversation:
    @classmethod
    def load(cls, llm: ChatOpenAI, cache_path: str = CONVERSATION_CACHE, max_token_limit: int = 1000) -> ConversationSummaryBufferMemory:
        Out.blue(f"Loading conversation history from {cache_path}")
        memory = ConversationSummaryBufferMemory(llm=llm, max_token_limit=max_token_limit, memory_key="chat_history", return_messages=True)
        if os.path.exists(cache_path):
            with open(cache_path, "rb") as f:
                memory.chat_memory = pickle.load(f)
        return memory

    @classmethod
    def save(cls, memory: ConversationSummaryBufferMemory, cache_path: str = CONVERSATION_CACHE) -> None:
        Out.blue(f"Saving conversation history to {cache_path}")
        with open(cache_path, "wb") as f:
            pickle.dump(memory.chat_memory, f)
        Out.green("Conversation history saved successfully")
```

Fig. 6: Clase Conversation, responsable de mantener la conversación con el usuario en memoria.

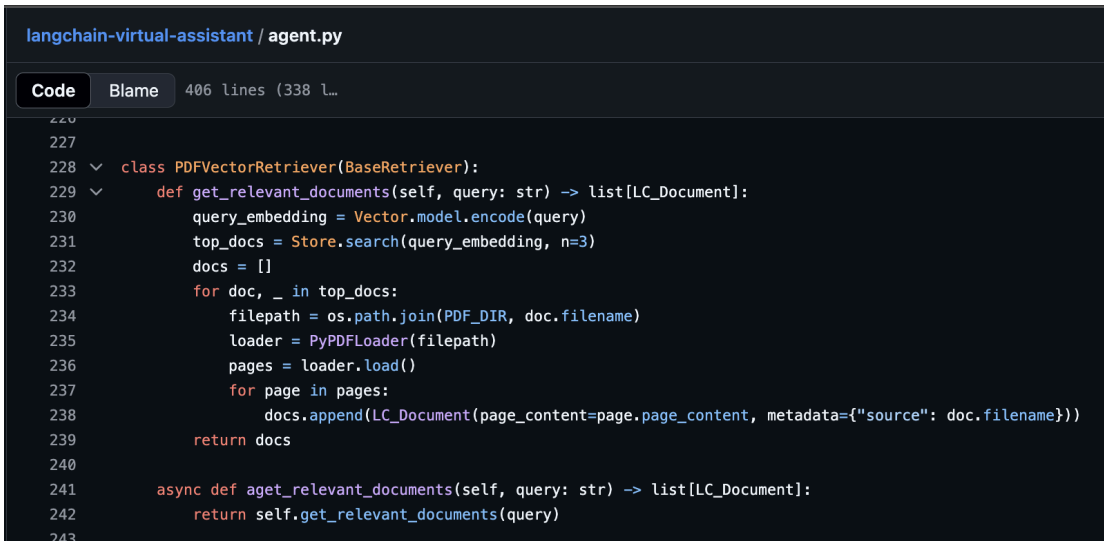

Fig. 7: Clase PDFVectorRetriever, responsable de cargar los PDFs.

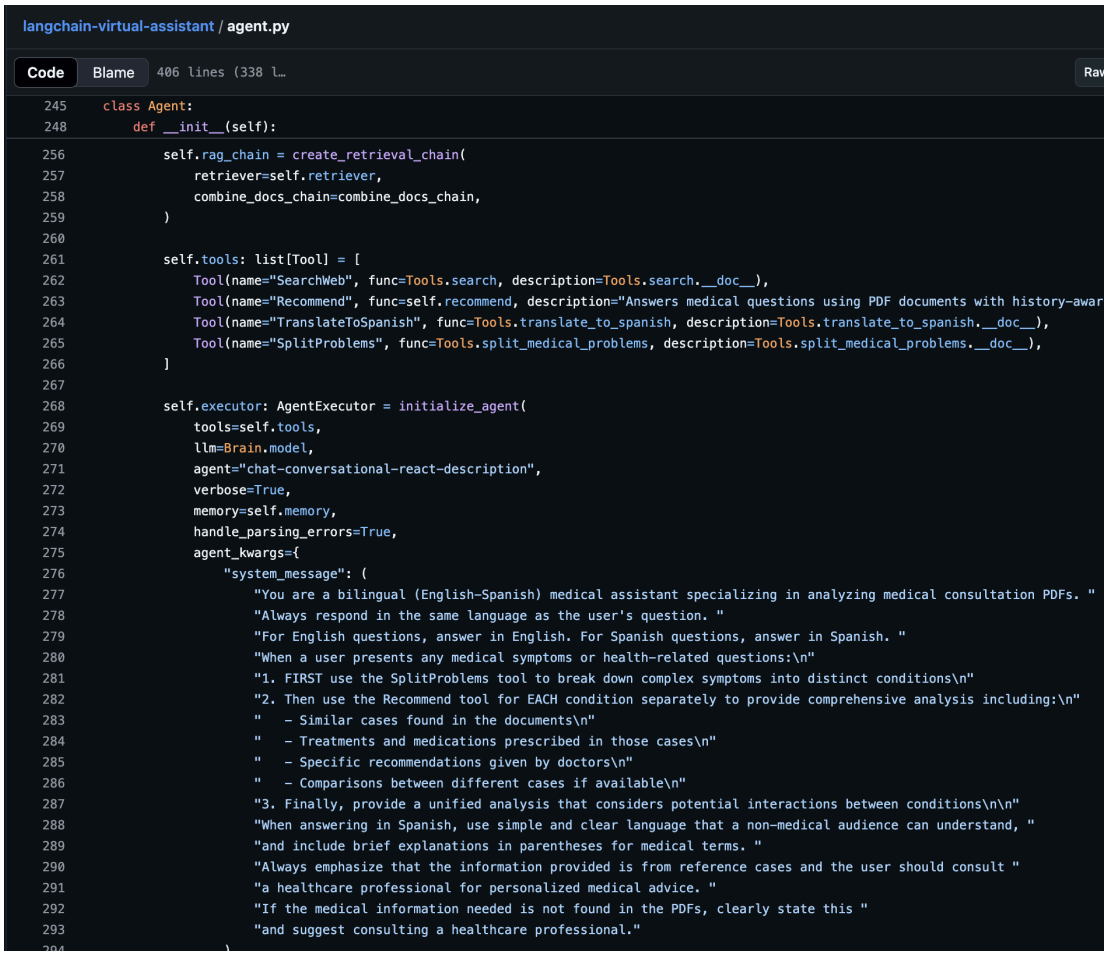

Fig. 8: Clase Agent, responsable de implementar el agente de LangChain.

langchain-virtual-assistant / agent.py

Code | Blame | 406 lines (338 l…

```python

class Tools:
    @staticmethod
    def search(query: str) -> str:
        """
        Performs a web search using Google and returns the first 2000 characters of the text content.
        Useful for finding current information or topics not covered in the internal knowledge or documents.
        Input must be a search query string.
        """
        url = f"https://www.google.com/search?q={query}"
        headers = {"User-Agent": "Mozilla/5.0"}
        res = requests.get(url, headers=headers, timeout=10)
        res.raise_for_status()
        soup = BeautifulSoup(res.text, "html.parser")
        main_content = soup.find("body")
        text = main_content.get_text(separator=" ", strip=True) if main_content else soup.get_text(separator=" ", strip=True)
        return text[:2000] if text else "No content found."

    @staticmethod
    def split_medical_problems(symptoms: str) -> str:
        """
        Splits complex medical symptoms or conditions into distinct problems for separate analysis.
        Returns a JSON array of conditions with related symptoms and search queries.
        Input must be the patient's symptoms or medical problems.
        """
        assert Brain.model, "Brain not initialized. Cannot perform problem splitting."
        chain = LLMChain(llm=Brain.model, prompt=Template.SPLIT_PROBLEMS)
        return chain.run(symptoms=symptoms)

    @staticmethod
    def extract_pdf_text(filename: str) -> str:
        """
        Extracts and returns the text content of a specific PDF file given its filename using PyPDFLoader.
        The filename must exactly match one of the files listed by `list_pdfs`.
        Input must be the filename.
        Returns the full text content of the PDF.
        """
        filepath = os.path.join(PDF_DIR, filename)
        assert os.path.exists(filepath), f"Error: PDF file '{filename}' not found in directory '{PDF_DIR}'."

        loader = PyPDFLoader(filepath)
        docs = loader.load()
        assert docs, f"No content loaded from {filename} using PyPDFLoader."

        text = "\n".join(doc.page_content for doc in docs)
        return " ".join(text.split())
```

Fig. 9: Clase Tools, responsable de extender el funcionamiento del agente.

```
>>> poetry run agent.py
¡Bienvenido! Soy su asistente médico virtual. ¿En qué puedo ayudarle hoy?
Type one of adiós, bye, chau, exit, goodbye, quit, salir to end.

# Prueba 1
>>> tengo dolor de cabeza
: Agent: Catalina Ríos fue la paciente con la queja de dolor de cabeza. Esta información se encuentra
en el Informe de Consulta Médica de Catalina Ríos, con fecha 02/05/2024.

# Prueba 2
>>> tengo dolor de hombro
Agent: El paciente con dolor de hombro tiene un diagnóstico de Síndrome del intestino irritable (SII)
y se le recomienda programar una ecografía para descartar complicaciones. Este diagnóstico y
recomendación se encuentran en el informe de consulta médica de Daniel Ortega en el archivo
'Daniel_Ortega_07-06-2024.pdf'.

# Prueba 3
>>> tengo ansiedad y depresión
Agent: El paciente con ansiedad y depresión puede encontrar información relevante en los archivos
'Sophia_Ortega_18-03-2025.pdf', 'Valentina_Ríos_14-09-2024.pdf', 'William_Ríos_20-01-2025.pdf',
entre otros.
```

Fig. 10: Casos de prueba del Asistente Virtual

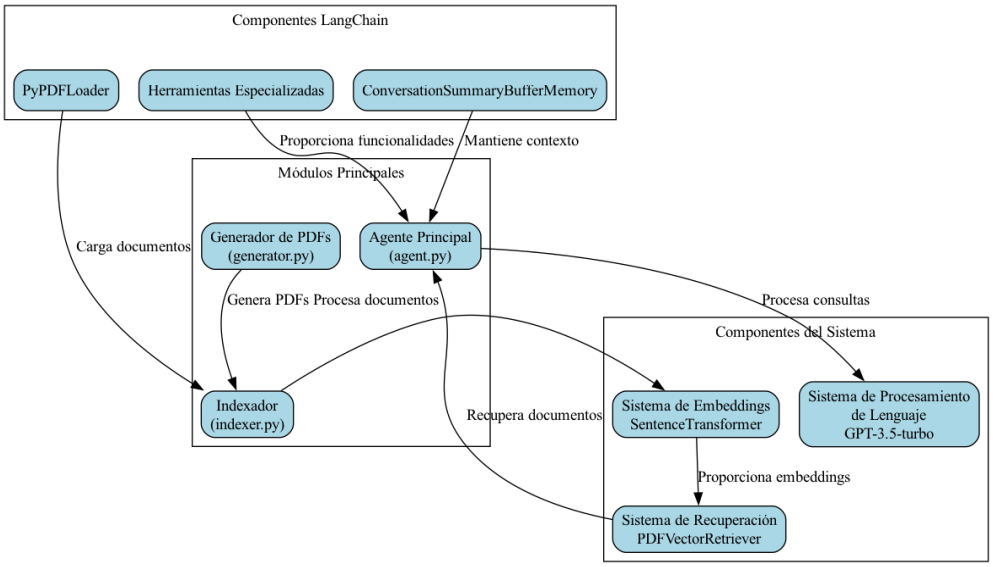

Fig. 11: Diagrama que describe la arquitectura general del sistema