1. Gestión de Estado y Memoria del Asistente

Para dotar al asistente de capacidad conversacional y persistencia durante la sesión del usuario, se ha implementado un módulo de gestión de memoria en RAM. Al tratarse de un prototipo que requiere baja latencia, hemos optado por una estructura de datos volátil en lugar de una base de datos relacional externa.

1.1. Estructura de Datos (Session Store)

La arquitectura de datos se basa en un diccionario global (_store) que actúa como almacén de sesiones. Cada usuario se identifica mediante un session_id, el cual mapea a una estructura de datos dividida en cuatro contextos lógicos:

History: Almacena la conversación cruda para dar contexto al LLM.

Memory: Almacena variables estructuradas (Destino, Duración, Estilo) necesarias para la lógica de negocio.

Pending: Gestiona estados intermedios o máquinas de estados (ej: esperando confirmación de una ambigüedad).

Itinerary: Caché del último resultado JSON válido generado.

```python
# Almacenamiento global en RAM
_store = {}

def get_session_data(session_id: str):
    # Inicialización Lazy: Si la sesión no existe, se crea con la estructura base
    if session_id not in _store:
        _store[session_id] = {
            "history": [],
            "memory": {"destination": "", "duration": "", "style": ""},
            "pending": {},
            "itinerary": None,
        }
    return _store[session_id]
```

1.2. Integración con LangChain (Tipado de Mensajes)

Para garantizar la compatibilidad con el motor de inferencia (LLM), el historial no se guarda como texto plano. Se utilizan las clases HumanMessage y AIMessage de la librería LangChain. Esto permite inyectar el historial formateado correctamente en los prompts del sistema, diferenciando claramente entre la entrada del usuario y la respuesta de la IA.

```python
from langchain_core.messages import HumanMessage, AIMessage

def add_message_to_history(session_id: str, role: str, content: str):
    data = get_session_data(session_id)
    if role == "user":
        data["history"].append(HumanMessage(content=content)) # Mensaj
    elif role == "ai":
        data["history"].append(AIMessage(content=content))    # Respue
```

1.3. Separación de Contexto Semántico

Un aspecto clave de la implementación es la separación entre el "chat" y los "datos del viaje". La función update_trip_memory permite actualizar selectivamente los parámetros que definen el itinerario (destination, duration, style) sin necesidad de procesar todo el historial de texto. Esto asegura que, independientemente de lo larga que sea la conversación, los datos críticos para la generación del JSON se mantengan limpios y accesibles.

```python
def update_trip_memory(session_id: str, dest=None, dur=None, style=None):
    """Actualiza selectivamente los parámetros fijos del viaje."""
    data = get_session_data(session_id)
    if dest: data["memory"]["destination"] = dest
    if dur: data["memory"]["duration"] = dur
    if style: data["memory"]["style"] = style
```

2. Motor de Inferencia e Integración de LLMs (llm_engine.py)

Este módulo actúa como una factoría de modelos que centraliza la conexión con los proveedores de IA. Su función principal es instanciar la clase correcta de LangChain (ChatGroq o OllamaLLM) basándose en la selección del usuario, devolviendo tanto el objeto funcional como un identificador del proveedor.

2.1. Selección de Estrategia Híbrida

El sistema evalúa el parámetro model_name para decidir si la inferencia se realiza en la nube (Groq) o en local (Ollama). Se han definido perfiles de rendimiento específicos: "fast" (Llama 3.1 8B), "smart" (Llama 3.3 70B) y "local".

```python
def get_chat_model(model_name: str = "smart"):
    # Estrategia para modelo Rápido (Nube)
    if model_name == "fast":
        llm = ChatGroq(model="llama-3.1-8b-instant", ...)
        return llm, "groq_8b"

    # Estrategia para modelo Local (Privacidad)
    if model_name == "local":
        llm = _fallback_local() # Instancia Ollama
        return llm, "ollama_local"
```

2.2. Control de Errores y Seguridad (Fail Fast)

Para evitar errores en tiempo de ejecución silenciosos, el código verifica explícitamente la existencia de la variable de entorno GROQ_API_KEY antes de intentar conectar con la nube. Si la clave no existe, se lanza una excepción bloqueante (RuntimeError) informando al desarrollador, en lugar de intentar la conexión y fallar después.

```python
    groq_api_key = os.getenv("GROQ_API_KEY")

    if model_name == "fast":
        if not groq_api_key:
            # Validación estricta antes de instanciar
            raise RuntimeError("GROQ_API_KEY no encontrada...")
```

2.3. Retorno de Tupla para la Interfaz

La función no retorna únicamente el modelo de lenguaje. Devuelve una tupla (llm_instance, provider_name). Este diseño permite que el Backend utilice la instancia llm para procesar el texto, mientras envía el provider_name (ej: "groq_70b") al Frontend para mostrar al usuario qué tecnología específica le está respondiendo.

```python
    # Retorno dual: Objeto funcional + Metadato
    return llm, "groq_70b"
```

3. Controlador de Diálogo y Orquestación (chat.py)

Este módulo actúa como el orquestador principal del Backend. Su responsabilidad es recibir la entrada del usuario, enriquecerla con contexto (Memoria o RAG), decidir la intención del usuario y construir el prompt adecuado para el LLM.

3.1. Extracción Heurística de Entidades (NLU Ligero)

Antes de invocar a la Inteligencia Artificial (que es costosa computacionalmente), el sistema realiza un pre-procesamiento del texto del usuario mediante Expresiones Regulares (regex). Esto permite detectar determinísticamente el Destino, la Duración y el Estilo del viaje si están presentes explícitamente, actualizando la memoria inmediatamente.

```
def parse_user_message(text: str) -> dict:
    # Extracción de duración (ej: "5 días")
    m = re.search(r"(\d+)\s*d[ií]a", text.lower())
    if m: out["duration"] = m.group(1) + " días"

    # Detección de Destino mediante lista controlada y patrón fallback
    if not out["destination"]:
        m2 = re.search(r"\bviaje a ([A-Za-zÁÉÍÓÚÑáéíóúñ]+)\b", text.lower())
        if m2: out["destination"] = m2.group(1).title()
```

3.2. Máquina de Estados (Fases del Viaje)

El controlador implementa una lógica condicional para determinar la FASE_ACTUAL de la conversación. Esta variable se inyecta en el prompt del sistema para modificar drásticamente el comportamiento de la IA:

Fase 1 (Perfilado): Faltan datos críticos -> La IA solo hace preguntas.

Fase 2 (Generación): Datos completos -> La IA genera el JSON inicial.

Fase 3 (Modificación): Ya existe itinerario -> La IA edita el JSON existente.

Fase 4 (Análisis): Se detecta contexto RAG/Archivos -> La IA integra información externa.

```
# Lógica de determinación de fase
if is_file_analysis:
    phase = 4
elif dest and dur:
    # Si ya hay itinerario, estamos modificando (Fase 3), si no, creando (Fase 2)
    phase = 3 if existing_itinerary else 2
else:
    phase = 1 # Faltan datos, solo conversar
```

3.3. Ingeniería de Prompt y Estructura JSON

Se utiliza un System Prompt robusto que define al asistente "Atlas". La clave de la implementación es la definición estricta del esquema de salida JSON. Se instruye al modelo para que, cuando esté en Fase 2 o

3, su respuesta sea exclusivamente un objeto JSON válido, sin texto introductorio ni conclusiones, facilitando el parsing posterior.

```
system_prompt = """
SIEMPRE recibirás una variable `FASE_ACTUAL`.
- FASE_ACTUAL = 2 o 3:
  La respuesta debe ser EXCLUSIVAMENTE un JSON válido.

### FORMATO JSON DEL ITINERARIO
{{
  "titulo": "...",
  "dias": [ ... ]
}}
"""
```

3.4. Integración RAG y Robustez en la Respuesta

El sistema integra un módulo opcional de RAG (Retrieval-Augmented Generation) para enriquecer el contexto con información de documentos externos si están disponibles. Además, incluye una capa de limpieza de la respuesta del LLM para eliminar bloques de código Markdown (```json) y asegurar que el Frontend reciba siempre un objeto procesable.

```
# Recuperación de contexto RAG si aplica
if rag_service and extra_info:
     retrieved = rag_service.retrieve_context(extra_info, session_id, k=3)

# Limpieza y Parseo de la respuesta
cleaned = response_text.replace("```json", "").replace("```", "").strip()
try:
    json_obj = json.loads(cleaned)
    return {"es_itinerario": True, **json_obj}
except:
    return {"es_itinerario": False, "mensaje_chat": cleaned}
```

4. Sistema de RAG y Análisis Multimodal (rag_handler.py)

Este módulo avanzado permite al usuario enriquecer el contexto del viaje subiendo archivos (PDFs de reservas, guías en TXT) o imágenes (fotos de lugares). El sistema implementa una arquitectura RAG (Retrieval-Augmented Generation) para que la IA pueda "leer" estos documentos antes de responder.

4.1. Análisis Multimodal (Visión + Texto)

El sistema es capaz de procesar diferentes tipos de entrada no estructurada:

Documentos de Texto (PDF/TXT/MD): Se utiliza PyMuPDFLoader y TextLoader de LangChain para extraer el texto plano.

Imágenes (JPG/PNG): Se integra un modelo de visión local (LLaVA ejecutado vía Ollama). La imagen se convierte a Base64 y se envía al modelo con un prompt específico para que describa la ubicación y el tipo de atracción turística.

```
def _analyze_image_with_ollama(self, file_path, filename):
    # Conversión a Base64 y llamada al modelo LLaVA
    prompt = "Analiza esta imagen como experto en turismo. Identifica ubicación...
    response = requests.post(..., json={"model": "llava", "images": [image_data]})
    return response.json().get("response")
```

4.2. Ingesta y Vectorización (ChromaDB)

Una vez extraído el texto (o el análisis de la imagen), este se divide en fragmentos (chunks) manejables utilizando RecursiveCharacterTextSplitter. Estos fragmentos se convierten en vectores matemáticos (embeddings) utilizando un modelo local (llama3.2:3b) y se almacenan en una base de datos vectorial persistente (ChromaDB). Esto permite búsquedas semánticas posteriores.

```
# Inicialización de la Base de Datos Vectorial
self.vector_store = Chroma(
    collection_name="trip_documents",
    embedding_function=OllamaEmbeddings(model="llama3.2:3b"),
    persist_directory="./chroma_db"
)

# Indexación del documento procesado
self.vector_store.add_documents(documents=splits)
```

4.3. Recuperación de Contexto (Retrieval)

Cuando el usuario hace una pregunta en el chat, el sistema no solo busca en la memoria de la conversación. La función retrieve_context consulta la base de datos vectorial para encontrar los fragmentos de documentos más relevantes (Similarity Search) asociados a la sesión actual. Este contexto recuperado se inyecta dinámicamente en el prompt del LLM, permitiéndole responder preguntas específicas sobre los archivos subidos.

```
def retrieve_context(self, query: str, session_id: str, k: int = 5):
    # Búsqueda semántica filtrada por usuario
    results = self.vector_store.similarity_search(
        query, k=k, filter={"session_id": session_id}
    )
    # Formateo del contexto para el LLM
    ctx = "📎 INFORMACIÓN DE ARCHIVOS ADJUNTOS:\n..."
    return ctx
```