EVENITY: Plataforma Inteligente para Gestión de Eventos

Melani Forsythe Matos C312
Olivia Ibañez Mustelier C311
Orlando De La Torre Leal C312

Facultad de Matemática y Computación, Universidad de La Habana, Cuba

1 Introducción

La planificación eficiente de actividades de ocio o profesionales requiere acceder a información confiable, actualizada y relevante sobre eventos disponibles. Sin embargo, esta información suele encontrarse dispersa en múltiples plataformas y en formatos heterogéneos. Este trabajo aborda dicha problemática mediante el diseño e implementación de un sistema automatizado de búsqueda y recomendación de eventos.

El sistema utiliza técnicas de inteligencia artificial distribuida, particularmente arquitecturas multiagente, para coordinar distintas tareas como la extracción de información, procesamiento semántico, enriquecimiento de datos, y generación de agendas. Además, incorpora mecanismos de razonamiento simbólico a través de grafos de conocimiento y búsqueda vectorial usando modelos de lenguaje.

2 Objetivos

- Desarrollar un sistema modular y extensible que permita la recolección, estandarización y recomendación de eventos culturales, deportivos y sociales.
- Utilizar una arquitectura multiagente para dividir las responsabilidades funcionales (scraping, procesamiento, embeddings, planificación, visualización).
- Aplicar técnicas de recuperación aumentada (RAG), razonamiento simbólico y optimización heurística.
- Presentar los resultados en una interfaz amigable que permita consultar eventos y generar agendas personalizadas.

3 Motivación y enfoque adoptado

Las arquitecturas multiagente ofrecen un paradigma flexible y robusto para el desarrollo de sistemas distribuidos. En este proyecto, se adoptó dicho enfoque para aislar responsabilidades, facilitar el desarrollo concurrente y permitir la evolución independiente de cada módulo. Cada agente opera de forma autónoma, se comunica mediante colas internas y puede escalarse de forma horizontal. El uso de modelos semánticos modernos permite interpretar las consultas del usuario en lenguaje natural, mientras que el grafo de conocimiento actúa como mecanismo de razonamiento sobre los eventos disponibles, aumentando así la relevancia de los resultados.

3.1 Módulo `crawler.py` – Extracción masiva de eventos desde APIs externas

El módulo `crawler.py` define la clase `EventScraper`, responsable de recolectar datos de eventos desde múltiples plataformas externas a través de sus APIs: `PredictHQ`, `SeatGeek` y `Ticketmaster`.

La arquitectura implementa procesamiento concurrente mediante `ThreadPoolExecutor` para maximizar el rendimiento en la descarga, transformación y almacenamiento de eventos.

Objetivos

- Descargar miles de eventos culturales, deportivos y sociales de distintas fuentes públicas.
- Estandarizar parcialmente los datos y almacenarlos como archivos individuales JSON.
- Operar de manera eficiente mediante paralelismo y control de errores.

Estructura de la clase `EventScraper`

- `scrape_seatgeek_events()`
- `scrape_predicthq_events()`
- `scrape_ticketmaster_events()`
- `process_<fuente>_event()`: funciones para normalización ligera.
- `run_all_scrapers()`: ejecuta las tres fuentes consecutivamente.

Funcionamiento técnico

Cada función de scraping realiza peticiones HTTP paginadas a la API correspondiente, respetando límites de eventos y pausas entre llamadas para evitar bloqueo por exceso de tráfico.

La respuesta JSON de cada API es adaptada a un formato intermedio común que incluye campos como:

- `id, title/name, date/time, location, venue, category, performers`.

Los resultados son guardados en la carpeta `eventos_completos` con nombre codificado por fuente e ID.

Manejo de errores y robustez

- Gestión de códigos de error HTTP.
- Reintentos configurables y registros detallados en logs.
- Separación entre descarga y almacenamiento para evitar corrupción de datos.

Escalabilidad

La clase permite configurar:

- Máximo número de eventos por fuente.
- Nivel de concurrencia (`max_workers`).
- Demora entre llamadas para cumplir con políticas de uso.

Conclusión: `crawler.py` constituye la puerta de entrada de datos al sistema. Su diseño concurrente y tolerante a errores permite recolectar de forma fiable una base de eventos suficientemente rica para alimentar los módulos posteriores de análisis, recomendación y visualización.

3.2 Módulo `scraping_incremental.py` – Detección de cambios mediante hashing

Este módulo implementa un sistema de control de versiones para eventos utilizando hashes MD5. Permite identificar automáticamente qué archivos JSON han sido añadidos o modificados desde la última ejecución del scraping, evitando reprocesar innecesariamente eventos no alterados.

Motivación

Dado el alto volumen de eventos procesados (miles por ejecución), reprocesarlos todos cada vez sería ineficiente y costoso. Este módulo permite aplicar un enfoque incremental, optimizando el tiempo y los recursos.

Estructura funcional

- `hash_file(path)`: genera un hash MD5 del contenido del archivo.
- `load_hashes()`: carga el diccionario de hashes previos desde `hashes.json`.
- `save_hashes(hashes)`: guarda los nuevos hashes actualizados.
- `archivos_a_procesar(folder)`: compara los archivos actuales con los hash previos y retorna solo los modificados o nuevos.

Formato de persistencia

Los hashes se almacenan en un archivo JSON persistente:

```
{
  "evento_1.json": "af12345abc...",
  "evento_2.json": "87ef45f1bb..."
}
```

Listing 1.1: Ejemplo de hashes.json

Integración con el sistema

Este módulo es utilizado directamente por el actualizador incremental para determinar:

- Qué eventos necesitan ser re-procesados (`procesamiento.py`).
- Qué embeddings deben regenerarse (`embedding.py`).

Ventajas

- Minimiza el tiempo de ejecución de procesos en lote.
- Evita el uso innecesario de recursos computacionales.
- Es completamente agnóstico a la fuente de eventos.

Conclusión: este módulo proporciona una solución ligera y efectiva para actualizar únicamente los eventos que realmente han cambiado, habilitando la eficiencia en sistemas de scraping y procesamiento en tiempo real o cuasi-real.

3.3 Módulo `actualizador_dinamico.py` – Actualización dinámica de eventos y embeddings

Este módulo permite mantener el sistema sincronizado con los datos más recientes disponibles en las APIs originales (SeatGeek, PredictHQ, Ticketmaster). Detecta eventos obsoletos o modificados, actualiza sus versiones, y regenera únicamente los embeddings necesarios.

Objetivo

Reducir al mínimo el reprocesamiento masivo de eventos, garantizando que la información ofrecida al usuario sea actual y completa.

Flujo de trabajo

1. Se detectan archivos candidatos mediante hashes (módulo `scraping_incremental.py`).
2. Se verifica si los eventos están incompletos, expirados o modificados.
3. Si es necesario, se vuelve a consultar la fuente original por ID.
4. Se reprocesa el evento con `EventProcessor` y se guarda.
5. Se regeneran los embeddings del evento actualizado de forma selectiva.

Funciones principales

- `revisar_y_actualizar_eventos()`: orquesta el análisis, consulta y reescritura de eventos.
- `buscar_evento_por_id()`: realiza la consulta directa a la API correspondiente.
- `is_event_obsolete_or_incomplete()`: evalúa si el evento debe actualizarse.
- `incremental_reembed()`: actualiza sólo los embeddings de los eventos modificados.
- `actualizar_eventos_y_embeddings()`: punto de entrada global del módulo.

Lógica de obsolescencia

Un evento se considera desactualizado si:

- Le falta información clave (fecha, título, venue).
- Su fecha ya ha pasado.
- Su hash no coincide con la versión remota actual.

Relación con otros módulos

- Se apoya en `procesamiento.py` para estandarizar los eventos.
- Usa `EventScraper` de `crawler.py` para realizar las nuevas descargas.
- Llama a `EventEmbedder` para actualizar los vectores semánticos.

Conclusión: este módulo garantiza que los datos del sistema estén actualizados de manera eficiente, sin necesidad de reconstruir el sistema completo, lo cual lo hace esencial para entornos con datos que cambian frecuentemente.

3.4 Módulo `procesamiento.py` – Estandarización y enriquecimiento de eventos

Este módulo define la clase `EventProcessor`, encargada de transformar los eventos descargados en bruto desde múltiples fuentes (como PredictHQ, SeatGeek y Ticketmaster) en una estructura homogénea y enriquecida.

Este procesamiento es crucial para que los eventos puedan ser utilizados por los módulos posteriores de embeddings, grafo de conocimiento, visualización y recomendación.

Objetivo

Uniformar eventos con estructuras distintas, normalizar fechas y localizaciones, e identificar errores o inconsistencias para asegurar la calidad de los datos.

Funcionamiento general

El módulo detecta automáticamente la fuente del evento (basado en el nombre del archivo o metadata), y aplica una función de procesamiento específica. Cada función traduce los campos fuente-específicos al formato interno esperado por el sistema.

Listing 1.2: Estructura general del evento enriquecido

```
{
  "metadata": {...},
  "basic_info": {...},
  "classification": {...},
  "temporal_info": {...},
  "spatial_info": {...},
  "participants": {...},
  "external_references": {...},
  "raw_data_metadata": {...}
}
```

Funciones principales

- `process_event(raw_data, source)`: entrada principal del procesador.
- `_process_predicthq`, `_process_seatgeek`, `_process_ticketmaster`: funciones específicas por fuente.
- `_normalize_datetime()`, `_process_temporal_info()`: normalización robusta de fechas y zonas horarias.
- `_post_process_event()`: limpieza final de datos y preservación de campos relevantes.

Validación y control de calidad

El procesador verifica:

- Que todos los eventos tengan un título, una fecha de inicio y algún dato de ubicación.
- Que se genere un hash del evento crudo para detectar cambios futuros.
- Que se eliminen campos vacíos o nulos que puedan entorpecer la visualización o el embedding.

En caso de error o datos incompletos, el evento se descarta o se marca con error, incluyendo un ID de rastreo.

Ventajas

- Unifica los formatos dispares de las distintas APIs externas.
- Permite desacoplar la lógica de scraping de la lógica de análisis.
- Mejora la trazabilidad de los eventos gracias a su metadata enriquecida.

Conclusión: el módulo `procesamiento.py` es un componente de limpieza, validación y enriquecimiento indispensable para garantizar la integridad y homogeneidad del sistema.

3.5 Módulo `embedding.py` – Representación semántica y búsqueda vectorial

Este módulo define la clase `EventEmbedder`, que implementa toda la lógica para transformar eventos en vectores semánticos mediante modelos de lenguaje, construir índices para búsquedas vectoriales, y realizar recomendaciones personalizadas.

Utiliza la biblioteca `SentenceTransformers` para generar embeddings y `FAISS` como motor de búsqueda vectorial.

Motivación

Los motores de búsqueda tradicionales se basan en coincidencia exacta de palabras clave. En cambio, este módulo permite encontrar eventos similares en función de su significado utilizando modelos de lenguaje profundo.

Componentes principales

- `generate_embeddings(eventos)`: convierte eventos a texto y luego a vectores normalizados.
- `build_faiss_index()`: construye el índice `FAISS` para búsquedas semánticas.
- `search(query, city)`: realiza búsqueda semántica, filtrada por ciudad.
- `recommend_events(...)`: recomienda eventos por preferencias del usuario (fecha, categoría, etc).
- `add_new_events() / save_event_data()`: permite actualización incremental del índice sin reconstruirlo.

Flujo de trabajo interno

1. Se cargan los eventos normalizados desde la carpeta `eventos_mejorados`.
2. Se construye un vector por evento usando un modelo `all-MiniLM-L6-v2`.
3. Se organiza un índice FAISS (brute-force o con clustering).
4. Al consultar, se embebe el texto de búsqueda y se calcula su similitud con los eventos.
5. Opcionalmente, se aplican filtros por fecha, ciudad, popularidad o categoría.

Diseño técnico

El módulo implementa un patrón *singleton* para evitar recalcular embeddings innecesariamente. Internamente, mantiene estructuras de datos:

- `self.eventos`: lista de eventos enriquecidos.
- `self.embeddings`: matriz NumPy con los vectores.
- `self.index`: índice `FAISS`.
- `self.metadata`: información auxiliar como ID, ciudad, fecha.

Ventajas

- Permite búsquedas flexibles como: *"eventos de rock este fin de semana"* o *"conciertos cerca de Madrid"*.
- Reduce significativamente el esfuerzo de los usuarios para encontrar contenido relevante.
- Aumenta la calidad de las recomendaciones al combinar semántica + metadatos.

Conclusión: este módulo constituye el núcleo del sistema de recomendación, al ofrecer capacidades semánticas avanzadas que mejoran tanto la exploración libre como la generación de agendas personalizadas.

3.6 Módulo `grafo_conocimiento.py` – Grafo de conocimiento entre eventos

Este módulo construye un grafo dirigido y etiquetado que representa las relaciones semánticas y estructurales entre los eventos del sistema. El grafo actúa como un mecanismo de razonamiento simbólico que complementa la búsqueda por embeddings, permitiendo relacionar eventos por ciudad, categoría, participantes, fecha y similitud.

Se utiliza la biblioteca `networkx` para construir y exportar el grafo en formatos manipulables (como `.graphml`) para su visualización y análisis.

Objetivo

Representar de manera explícita y navegable las conexiones entre eventos para:

- Detectar comunidades temáticas (conciertos, festivales, deportes).
- Facilitar recomendaciones basadas en estructura de red.
- Enriquecer el sistema con capacidades de inferencia simbólica.

Estructura del grafo

Los nodos representan eventos únicos. Los arcos representan relaciones del tipo:

- `ocurre_en`: conecta un evento con su ciudad.
- `de_categoria`: conecta un evento con su categoría (música, deporte...).
- `realizado_por`: conecta un evento con un artista o grupo participante.
- `similar_a`: conecta eventos semánticamente similares.

Funciones principales

- `construir_grafo_de_conocimiento()`: función principal que itera sobre todos los eventos y genera nodos y aristas con atributos.
- `agregar_relaciones()`: subrutinas que conectan eventos entre sí o con nodos abstractos (categoría, ciudad...).
- `exportar_grafo(nombre)`: guarda el grafo en disco como archivo GraphML para visualización externa.

Ventajas del uso de grafos

- Permite visualizar y analizar relaciones ocultas entre eventos.
- Facilita razonamientos como "eventos relacionados con este artista" o "eventos similares en otra ciudad".
- Integra búsqueda simbólica con búsqueda semántica (complementariedad).

Aplicaciones futuras

- Integración con motores de razonamiento (e.g., motores de inferencia RDF).
- Recomendaciones contextuales basadas en caminatas aleatorias sobre el grafo.
- Visualizaciones dinámicas en la interfaz web del usuario.

Conclusión: el módulo `grafo_conocimiento.py` introduce una capa de representación estructurada del conocimiento que potencia las capacidades de descubrimiento y recomendación del sistema más allá del simple matching textual o vectorial.

3.7 Módulo `optimizador.py` – Generación de agendas personalizadas

El módulo `optimizador.py` implementa la lógica de planificación que permite construir una agenda óptima de eventos, basada en múltiples criterios definidos por el usuario: fechas disponibles, ciudades de interés, categorías preferidas, número máximo de eventos, entre otros.

Para ello, se utiliza una técnica metaheurística basada en *Hill Climbing* (escalada de colina) con un sistema de evaluación multiobjetivo.

Motivación

Dado un conjunto de eventos y un conjunto de restricciones o preferencias, el problema de seleccionar los eventos más relevantes es un problema combinatorio con múltiples variables. No se trata sólo de maximizar popularidad, sino de optimizar un balance entre:

- **Diversidad temática**
- **Proximidad temporal y geográfica**
- **No solapamiento**
- **Preferencias explícitas del usuario**

Funciones principales

- `filtrar_eventos_por_rango()`: prefiltra los eventos según las fechas deseadas.
- `obtener_eventos_optimales()`: función principal que implementa la heurística de optimización.
- `evaluar_agenda()`: evalúa una combinación de eventos según la función objetivo ponderada.
- `agendas_compatibles()`: verifica que los eventos seleccionados no se solapen temporalmente.

Función objetivo

La puntuación de una agenda se calcula como:

$$\text{Score} = \alpha \cdot \text{popularidad} + \beta \cdot \text{diversidad} + \gamma \cdot \text{coherencia temporal} + \delta \cdot \text{ajuste a preferencias}$$

Los pesos $\alpha, \beta, \gamma, \delta$ pueden ajustarse dinámicamente según el contexto de usuario.

Algoritmo utilizado

1. Se parte de una agenda inicial aleatoria.
2. En cada iteración se intercambia un evento por otro y se evalúa la nueva agenda.
3. Si la agenda mejora, se acepta el cambio.
4. El proceso se repite hasta llegar a un número máximo de iteraciones sin mejora.

Restricciones manejadas

- No solapamiento de horarios entre eventos.
- Número máximo de eventos definidos por el usuario.
- Fecha mínima y máxima de búsqueda.
- Ciudades y categorías seleccionadas.

Ventajas

- Genera resultados personalizados y factibles.
- Permite adaptarse fácilmente a nuevos criterios (por ejemplo, presupuesto o distancia).
- Brinda recomendaciones accionables, listas para mostrar en la interfaz.

Conclusión: el módulo `optimizador.py` es el componente responsable de transformar una lista genérica de eventos en una agenda coherente, atractiva y personalizada para el usuario final, utilizando métodos heurísticos eficaces.

3.8 Módulo `sistema_multiagente.py` – Coordinador y orquestador de agentes

Este módulo define el funcionamiento del sistema multiagente que da vida al organizador inteligente de eventos. Cada componente funcional del sistema (scraper, procesador, embebedor, optimizador, etc.) se encapsula como un agente autónomo con su propia lógica, canal de comunicación y ciclo de vida.

La comunicación entre agentes se realiza mediante colas internas (bandejas) definidas en `contexto_global.py`. Cada agente escucha su bandeja y actúa en función de los mensajes recibidos.

Arquitectura general

- Cada agente se define como una subclase de `threading.Thread`.
- Al inicializar el sistema, se crean los hilos y se registran sus bandejas.
- Se utiliza un *message bus* implícito con colas internas para desacoplar los agentes.

Agentes definidos

- `AgenteScraper`: descarga eventos desde las APIs externas.
- `AgenteProcesador`: normaliza y enriquece los eventos brutos.
- `AgenteActualizador`: detecta cambios y mantiene la base de datos actualizada.

- `AgenteEmbedding`: genera representaciones vectoriales semánticas.
- `AgenteGrafo`: construye el grafo de conocimiento entre eventos.
- `AgenteOptimizador`: genera la mejor agenda personalizada.
- `AgenteBusquedaInteractiva`: responde a consultas del usuario vía API.
- `AgenteGapFallback`: busca eventos adicionales si la búsqueda retorna pocos resultados.

Ejecución concurrente

Cada agente corre en un hilo propio, lo cual permite que:

- Varias tareas ocurran en paralelo (scraping, embedding, visualización).
- El sistema escale horizontalmente agregando más agentes.
- Se mantenga la reactividad del sistema frente a múltiples usuarios o solicitudes.

Gestión de mensajes

Los agentes reciben mensajes con comandos específicos, por ejemplo:

- `"scrapear"` → inicia descarga de eventos.
- `"procesar"` → transforma un archivo descargado.
- `"buscar"` → consulta eventos en el índice semántico.
- `"generar_agenda"` → devuelve una agenda personalizada.

Los mensajes tienen formato tipo diccionario y pueden incluir datos adicionales (rango de fechas, filtros, preferencias).

Ventajas del enfoque multiagente

- Desacoplamiento entre funcionalidades.
- Fácil extensión e integración de nuevos servicios (agentes).
- Robustez frente a fallos individuales (cada agente puede reiniciarse).
- Natural paralelismo y distribución de tareas.

Conclusión: el módulo `sistema_multiagente.py` implementa el corazón de la arquitectura distribuida del sistema. Gracias a este diseño, los módulos funcionan como entidades autónomas coordinadas, aumentando la flexibilidad, escalabilidad y resiliencia de toda la solución.

3.9 Módulo `lanzar_api.py` – Lanzador de la API REST

El módulo `lanzar_api.py` funciona como punto de arranque del sistema, encargándose de poner en marcha el servidor web que permite acceder a los servicios de la API.

Se basa en la instancia de `Flask` definida en el módulo `servidor_base.py`, e importa todas las rutas definidas en `servidor_api.py` para exponerlas al exterior.

Listing 1.3: Lanzamiento del servidor API

```
from api.servidor_base import app, logger

def iniciar_api():
    import api.servidor_api
    logger.info("␣API␣iniciada␣en␣http://localhost:8502")
    app.run(host="0.0.0.0", port=8502)
```

Funcionamiento detallado

- `from api.servidor_base import app`: reutiliza la instancia de Flask definida anteriormente.
- `import api.servidor_api`: importa las rutas y lógica de negocio al servidor.
- `app.run(...)` : inicia el servidor en el puerto 8502 y lo expone a todas las interfaces de red.

Contexto de ejecución

Este script está pensado para ser ejecutado como proceso independiente (por ejemplo, desde un hilo o script principal del sistema), y es útil tanto para despliegues locales como remotos. Puede integrarse con procesos en segundo plano, gestores como `supervisord` o servicios en la nube.

Conclusión: `lanzar_api.py` encapsula el procedimiento de arranque del servidor, asegurando que toda la lógica esté cargada y disponible para recibir solicitudes HTTP.

3.10 Módulo `servidor_base.py` – Inicialización del servidor Flask

Este módulo se encarga de la inicialización y configuración básica de la API que sirve como interfaz entre el usuario (mediante la aplicación visual o llamadas externas) y el sistema de agentes.

Se utiliza el microframework `Flask` por su simplicidad, modularidad y compatibilidad con arquitecturas REST. Aquí se define la aplicación base y un endpoint mínimo de verificación.

Listing 1.4: Inicialización de la aplicación Flask

```
from flask import Flask
import logging

app = Flask(__name__)
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@app.route("/ping")
def ping():
    return "pong"
```

Componentes principales

- `app = Flask(__name__)`: crea una instancia de la aplicación web.
- `logger`: inicializa un sistema de logs para registrar eventos del servidor.
- `/ping`: endpoint REST que permite verificar si la API está activa; devuelve el mensaje `"pong"`.

Importancia en la arquitectura

Este módulo actúa como punto de entrada para la API REST. No contiene lógica de negocio, sino que configura la base sobre la que otros módulos (como `servidor_api.py`) montan sus rutas. También se reutiliza en el archivo `lanzar_api.py` para ejecutar el servidor.

Conclusión: este módulo representa el esqueleto del servidor web y permite una arquitectura limpia donde la lógica de negocio se define por separado, mejorando así la mantenibilidad y el desacoplamiento del sistema.

3.11 Módulo `servidor_api.py` – API REST para interacción externa

Este módulo define todos los endpoints públicos que permiten interactuar con el sistema organizador de eventos desde el exterior. Implementado con `Flask`, actúa como puerta de entrada HTTP para usuarios o servicios que deseen buscar eventos, generar agendas o consultar metadatos.

Integración con el sistema multiagente

El módulo no procesa directamente las solicitudes. En cambio, transmite las peticiones a los agentes correspondientes mediante colas internas (definidas en `contexto_global.py`) y espera las respuestas. Esto asegura un diseño desacoplado y asíncrono.

Endpoints implementados

- `/ping` (GET): Verificación básica de vida del sistema.
- `/ciudades` (GET): Lista de ciudades disponibles.
- `/categorias` (GET): Lista de categorías detectadas.
- `/buscar` (POST): Búsqueda semántica de eventos con filtros.
- `/agenda` (POST): Generación de agenda óptima basada en preferencias.

Formato de mensajes

Los endpoints POST aceptan y devuelven datos en formato JSON. Cada petición se traduce a un mensaje que se coloca en la bandeja del agente correspondiente, incluyendo:

- `tipo`: tipo de operación (ej. `"buscar"`, `"generar_agenda"`).
- `data`: parámetros específicos (rango de fechas, ciudad, número máximo, texto de consulta, etc.).

Listing 1.5: Ejemplo de envío de mensaje a un agente

```
bandeja = obtener_bandeja("AgenteBusquedaInteractiva")
respuesta = Queue()
bandeja.put({
    "tipo": "buscar",
    "data": parametros,
    "respuesta": respuesta
})
resultado = respuesta.get(timeout=25)
```

Ventajas del diseño

- Separación clara entre capa web (API) y lógica de negocio (agentes).
- Escalabilidad horizontal: múltiples instancias de API pueden compartir el mismo backend.
- Simplicidad de uso para el usuario final o integradores externos.

Conclusión: el módulo `servidor_api.py` permite exponer el sistema de organización de eventos a través de una API REST robusta y extensible, facilitando la consulta, la personalización y la integración con otras plataformas.

3.12 Módulo `contexto_global.py` – Gestión de colas de mensajes entre agentes

Este módulo define el sistema de comunicación interna entre los agentes del sistema multiagente. Se trata de una implementación basada en colas FIFO (`Queue`) que permite a cada agente enviar y recibir mensajes de manera asíncrona, desacoplada y segura.

El módulo actúa como un registro centralizado que asocia el nombre de cada agente a una cola compartida que opera como su "bandeja de entrada". Este diseño imita una arquitectura distribuida mediante un sistema de mensajería local.

Listing 1.6: Definición del sistema de registro de bandejas por agente

```
from queue import Queue

_bandejas_por_agente = {}

def registrar_bandeja(nombre_agente: str, bandeja: Queue):
    _bandejas_por_agente[nombre_agente] = bandeja

def obtener_bandeja(nombre_agente: str) -> Queue:
    return _bandejas_por_agente.get(nombre_agente)

def get_all_bandejas():
    return _bandejas_por_agente
```

Funciones principales

- `registrar_bandeja(nombre_agente, bandeja)`: permite registrar una cola asociada a un agente identificado por su nombre. Esta cola será usada por otros agentes para enviarle mensajes.
- `obtener_bandeja(nombre_agente)`: permite a un agente o controlador recuperar la cola de mensajes de otro agente. Devuelve la cola si está registrada, o `None` en caso contrario.
- `get_all_bandejas()`: retorna un diccionario completo con todas las bandejas registradas en el sistema, útil para tareas de monitoreo o inicialización.

Rol en la arquitectura del sistema

Este módulo es fundamental para la arquitectura multiagente porque actúa como middleware ligero de comunicación. Gracias a su uso:

- Los agentes no necesitan conocer la implementación interna de otros agentes para interactuar.
- Se facilita el paralelismo, ya que cada agente puede procesar su bandeja en un hilo o proceso separado.
- Se logra un sistema extensible: agregar nuevos agentes solo requiere registrar una nueva bandeja.

Conclusión: `contexto_global.py` es un componente esencial que habilita la orquestación entre módulos, garantizando modularidad, desacoplamiento y concurrencia controlada.

3.13 Módulo `visual.py` – Interfaz de usuario con Streamlit

Este módulo implementa la interfaz gráfica del sistema utilizando la biblioteca `Streamlit`, diseñada para construir aplicaciones web ligeras orientadas a ciencia de datos. A través de esta interfaz, el usuario puede buscar eventos y generar una agenda personalizada en función de sus preferencias.

Estructura general

El módulo organiza la interfaz en dos bloques principales:

- Búsqueda personalizada de eventos por texto, ciudad, fecha y categoría.
- Generación automática de agenda optimizada.

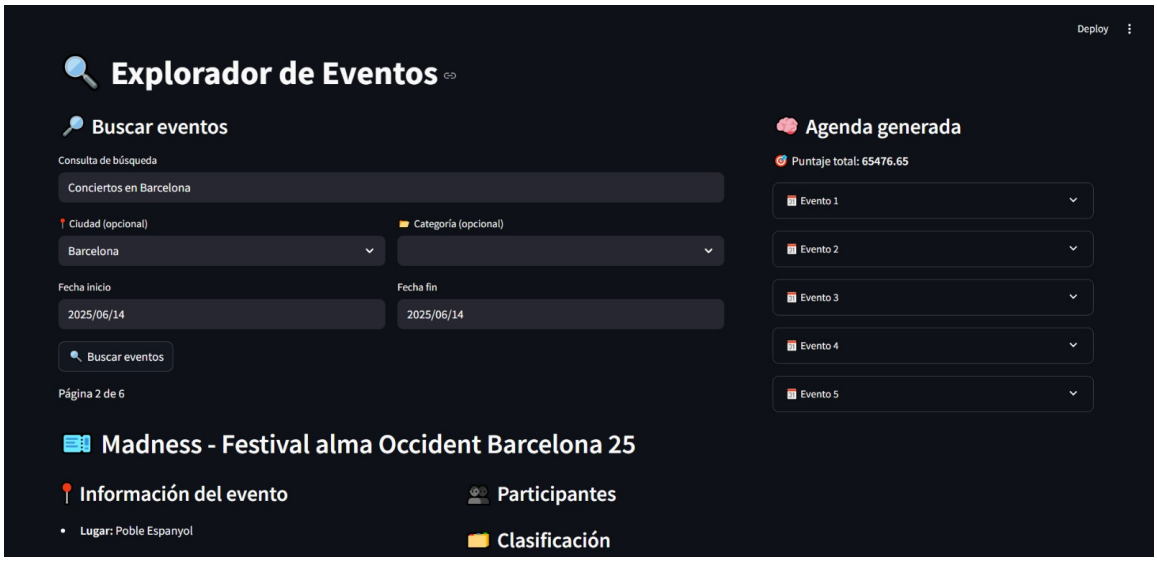

Fig. 1: Interfaz de usuario del sistema organizador de eventos

Componentes funcionales

- `st.text_input()`: entrada de búsqueda libre.
- `st.multiselect()`: filtros por ciudades y categorías.
- `st.date_input()`: rango de fechas.
- `st.number_input()`: número máximo de eventos para la agenda.
- `st.button()`: disparadores para buscar o generar agenda.

Visualización de resultados

Los eventos encontrados se muestran uno por uno con:

- Título
- Ciudad
- Fecha
- Artista, categoría u organizador (si están disponibles)

Además, se incluyen elementos visuales como separadores, espaciado y estructuras en columnas para organizar mejor el contenido.

Gestión de resultados

- Paginación: los resultados están limitados a 10 por página para evitar saturación visual.
- Agenda lateral: la agenda generada aparece en un panel separado para no sobreescribir los resultados de búsqueda.
- Personalización: el número de eventos en la agenda puede configurarse antes de su generación.

Comunicación con la API

Las acciones del usuario disparan peticiones POST a los endpoints REST definidos en `servidor_api.py`, mediante `requests.post()`, y muestran la respuesta formateada.

Ventajas

- Interfaz intuitiva sin necesidad de conocimientos técnicos.
- Facilidad para integrar nuevos campos o visualizaciones.
- Posibilidad de extender la interfaz con mapas, calendarios, o gráficos en tiempo real.

Conclusión: el módulo `visual.py` proporciona una puerta de entrada accesible y funcional al sistema, permitiendo a los usuarios explorar eventos y construir su agenda ideal de forma interactiva.

4 Diagrama de interaccion entre modulos

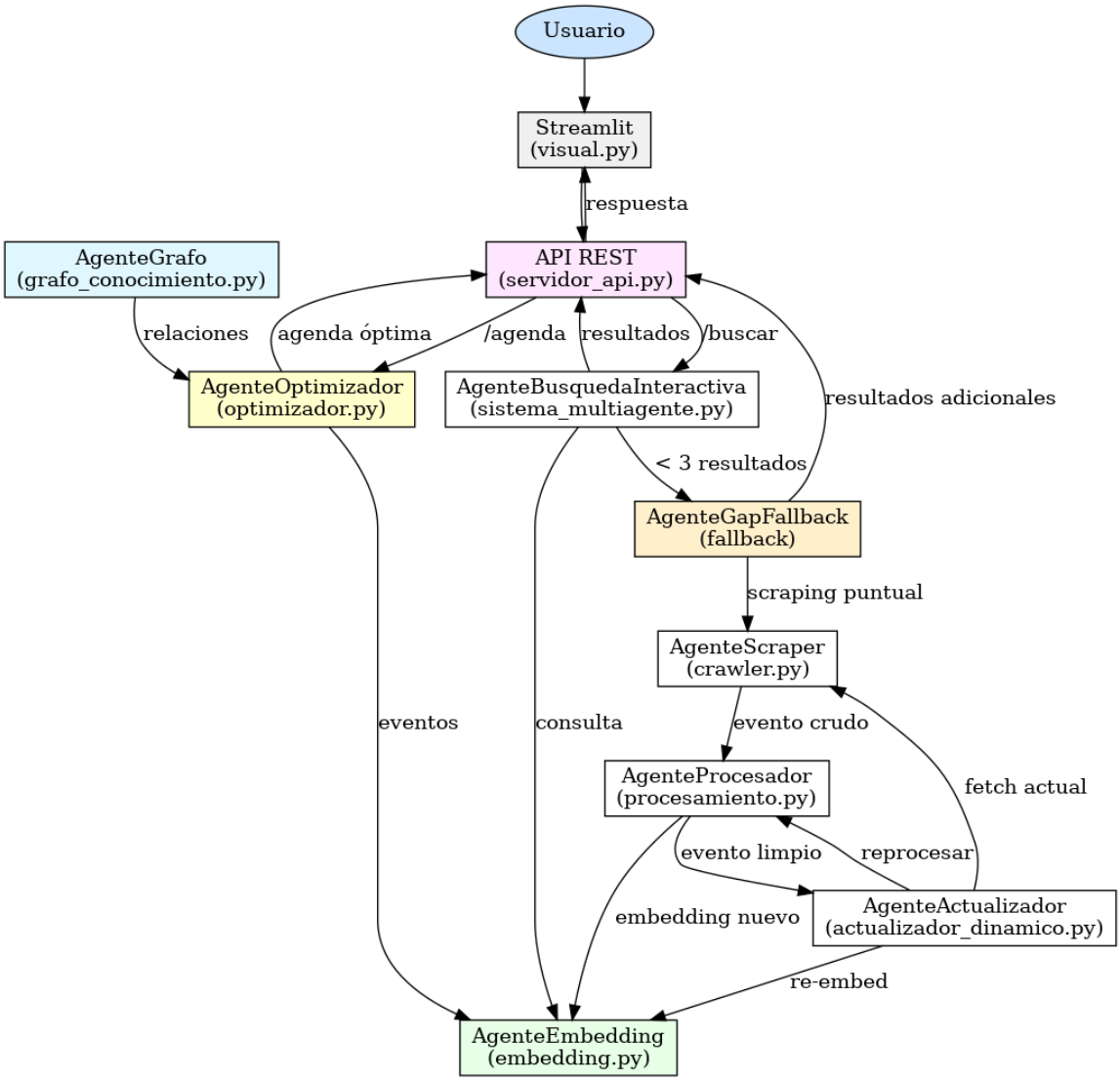

Fig. 2: Diagrama de interacción entre los módulos del sistema. El flujo parte del usuario a través de la interfaz en Streamlit y se comunica con el backend mediante una API REST. Los agentes colaboran para realizar búsqueda semántica, scraping, enriquecimiento, recomendaciones y planificación. El agente de *fallback* entra en acción si los resultados iniciales son insuficientes y también responde a la API.

5 Experimentación: Comparativa de Heurísticas y Funciones de Evaluación

Con el objetivo de validar la estrategia de planificación del sistema, se diseñó un conjunto de experimentos centrados en dos aspectos fundamentales: (1) la función heurística utilizada para evaluar agendas y (2) el algoritmo de optimización que recorre el espacio de soluciones.

5.1 Dos enfoques heurísticos: básica vs mejorada

Se compararon dos funciones heurísticas distintas para evaluar la calidad de una agenda de eventos:

- **Heurística básica**: se centra en coincidencias simples de ciudad, categoría y fecha, con penalizaciones mínimas por traslados largos y sin análisis de solapamientos ni diversidad.
- **Heurística mejorada (actual)**: incorpora dimensiones ponderadas (categoría, ciudad, duración, traslados, diversidad temática, huecos entre eventos y conectividad en el grafo de conocimiento). Aumenta la capacidad discriminativa y fomenta agendas realistas y balanceadas.

Table 1: Comparativa de heurísticas y algoritmos de optimización

Heurística	Algoritmo	Score Prom.	Mejor Score	Desv. Est.	Tiempo (s)	Iter.
Mejorada	Random	177,111.88	201,620.48	9,126.55	0.472	100
	Hill Climbing	204,594.38	233,331.67	10,889.31	0.471	101
	Simulated Annealing	203,944.83	227,875.27	9,853.22	0.435	91
Básica	Random	-1.00	-1.00	0.00	0.507	100
	Hill Climbing	-36,298.72	-20,701.62	5,836.42	0.475	101
	Simulated Annealing	-36,948.91	-20,796.40	6,125.82	0.439	91

Nota: Todos los valores numéricos representan métricas de rendimiento comparativas.

Este bloque muestra:

Que la heurística básica tiene resultados pobres, incluso negativos, especialmente con algoritmos como Hill Climbing.

Que la heurística mejorada se beneficia de forma clara al combinarse con Hill Climbing o Simulated Annealing, obteniendo puntuaciones mucho más altas, convergencia más rápida y menor variabilidad.

5.2 Heurísticas de búsqueda evaluadas

Se evaluaron tres estrategias para recorrer el espacio de soluciones:

Búsqueda Aleatoria

Técnica base sin retroalimentación. Resultó la peor estrategia:

- No genera mejoras sustanciales.
- Score promedio: **177111**, con alta dispersión y baja confiabilidad.

Ascenso de Colinas (Hill Climbing)

Estrategia local eficiente que mejora iterativamente:

- Score promedio: **204594**, mejor que Random.
- Problema: puede estancarse en óptimos locales.

Recocido Simulado (Simulated Annealing)

Explora soluciones peores con cierta probabilidad controlada por temperatura:

- Score promedio: **203945**, apenas por debajo de Hill Climbing.
- Mejor estabilidad: desviación estándar más baja (**9853**).
- Mejor comportamiento de convergencia (ver Fig. **??**).

5.3 Comparativas Gráficas

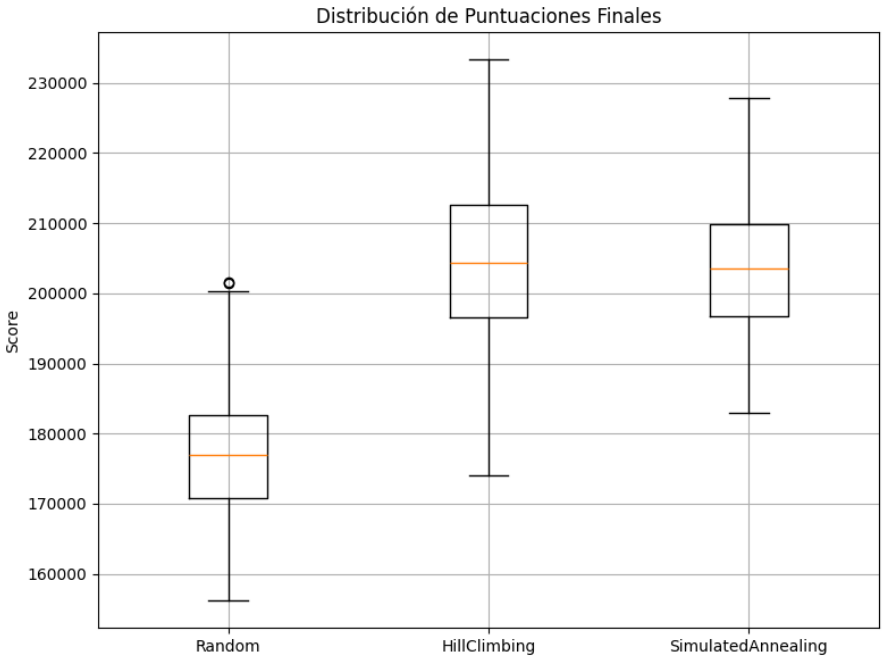

Fig. 3: Distribución de puntuaciones finales por heurística.

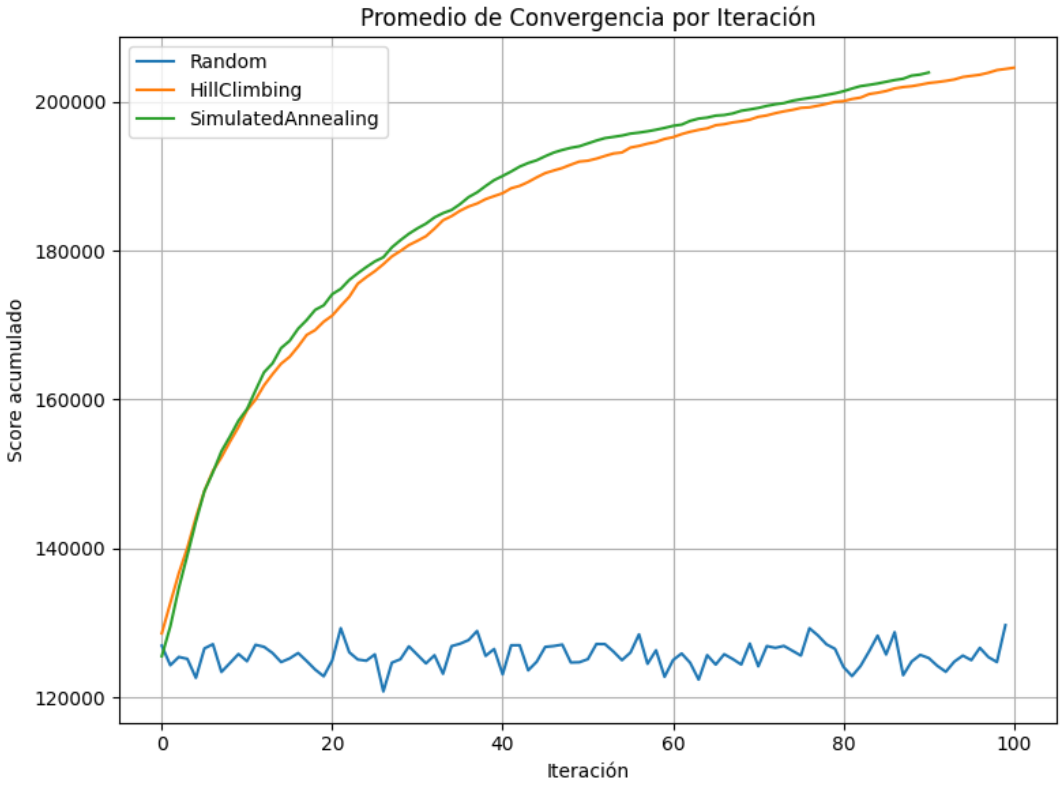

Fig. 4: Promedio de convergencia por iteración. Simulated Annealing supera a Hill Climbing desde la iteración 50.

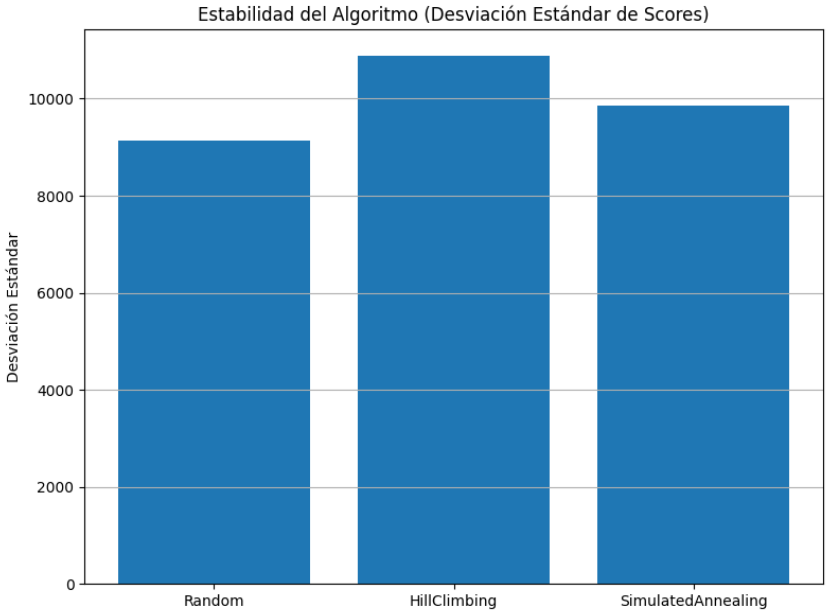

Fig. 5: Estabilidad de los algoritmos. Simulated Annealing mantiene una desviación más controlada.

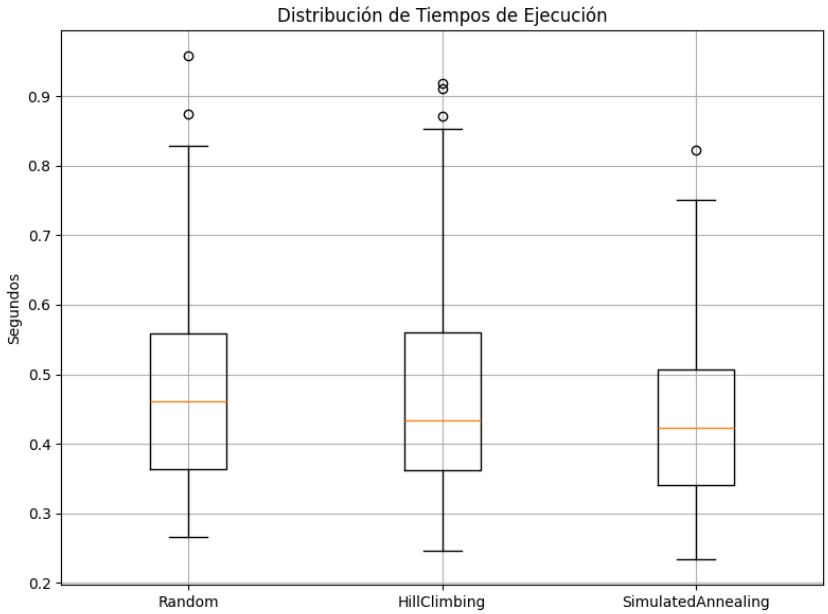

Fig. 6: Tiempos de ejecución. Todos completan en menos de 1 segundo.

5.4 Justificación de la elección: Simulated Annealing

Tras múltiples ejecuciones sobre 70 escenarios distintos, se concluye que **Simulated Annealing ofrece el mejor equilibrio entre rendimiento, estabilidad y exploración**:

- Supera a Random y es comparable a Hill Climbing en score absoluto.
- Tiene menor dispersión de resultados (desviación estándar más baja).
- Presenta la curva de convergencia más suave y consistente.
- Evita estancamientos en óptimos locales mediante su mecanismo probabilístico.

5.5 Conclusión

La combinación de una función de evaluación semánticamente enriquecida con un algoritmo exploratorio como Simulated Annealing permite generar agendas de eventos de alta calidad, con buen balance entre exploración y explotación. Esta decisión está respaldada por los resultados empíricos obtenidos, tanto en términos de precisión como de estabilidad.

6 Evaluación de Modelos de Embeddings para Recuperación Semántica

El sistema de recomendación de eventos se apoya en representaciones vectoriales generadas por modelos de *sentence embeddings*. Para determinar qué modelo proporciona mejores resultados en la tarea de recuperación de eventos relevantes, se evaluaron múltiples modelos preentrenados bajo distintas condiciones de consulta.

6.1 Metodología

Se definieron tres tipos de consultas simuladas:

- **con_fecha**: incluyen restricciones temporales explícitas.
- **estándar**: describen eventos en lenguaje natural sin restricción temporal.
- **naturales**: simulaciones más libres de preferencias del usuario.

Los modelos fueron evaluados con las métricas estándar de recuperación:

- **NDCG@10** (Discounted Cumulative Gain): mide la relevancia y orden de los eventos recomendados.
- **Precision@10**: proporción de eventos relevantes entre los 10 primeros resultados.
- **Recall@10**: proporción de eventos relevantes recuperados entre todos los relevantes.

6.2 Modelos evaluados

Entre los modelos comparados se encuentran:

- `all-MiniLM-L6-v2`, `all-MiniLM-L12-v2`
- `mpnet-base-v2`, `paraphrase-MiniLM`, `LaBSE`
- `msmarco-MiniLM-L6-cos-v5`, `msmarco-distilbert-base-v3`
- `distiluse-base-multilingual-cased`

Los modelos `paraphrase-multilingual-mpnet-base-v2` y `msmarco-distilbert-base-v3` sobresalen con mejores resultados globales en las tres métricas. Por el contrario, modelos ligeros como `MiniLM-L12-v2` o `paraphrase-MiniLM-L3-v2` ofrecen menor rendimiento aunque con tiempo de inferencia más rápido.

6.3 Análisis por tipo de consulta

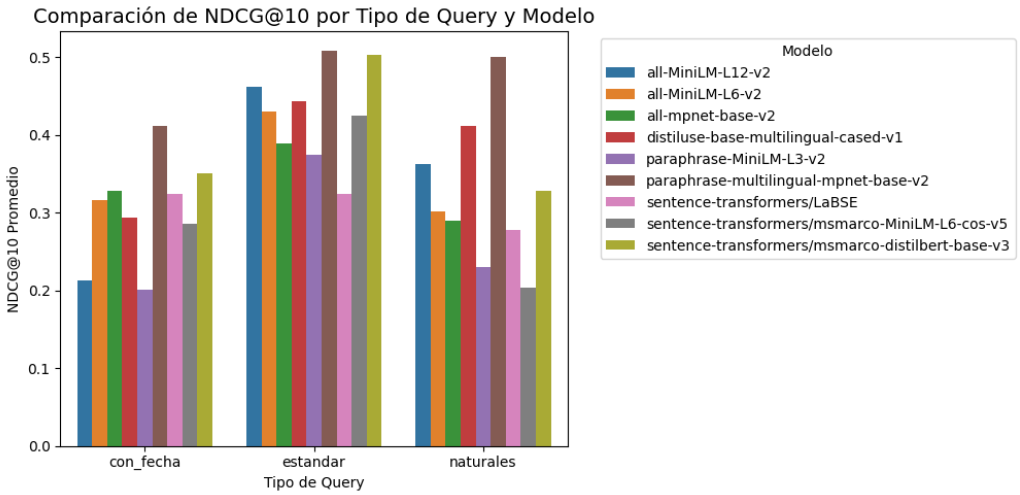

Fig. 7: Comparación de NDCG@10 por tipo de consulta y modelo.

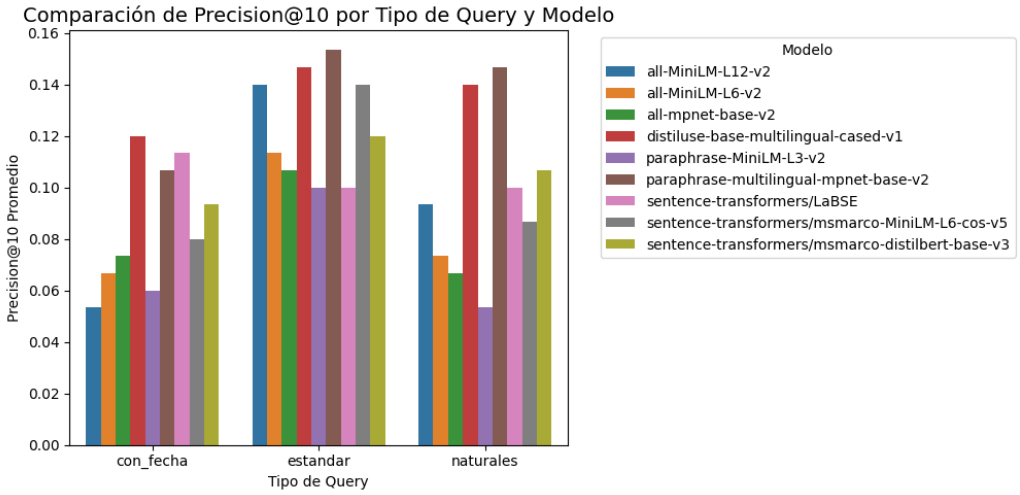

Fig. 8: Comparación de Precision@10 por tipo de consulta y modelo.

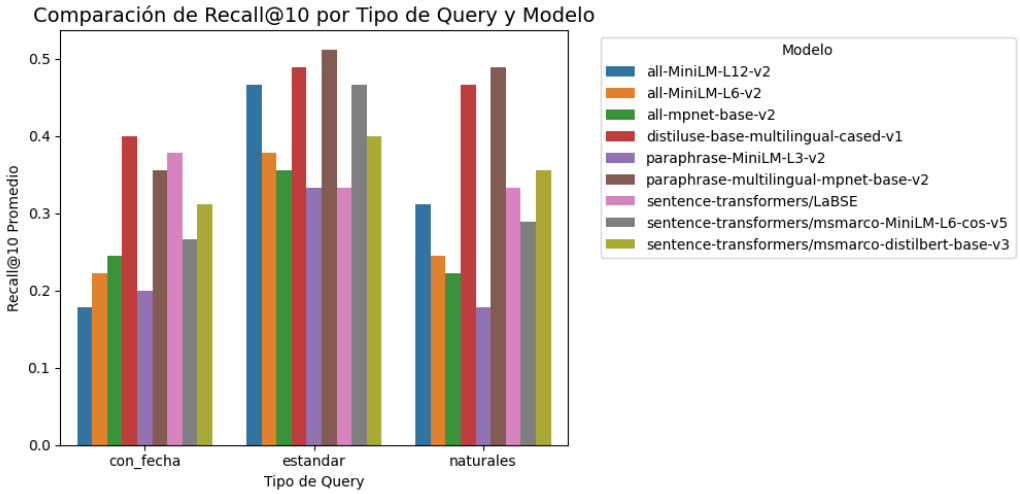

Fig. 9: Comparación de Recall@10 por tipo de consulta y modelo.

`msmarco` y `multilingual-mpnet-base-v2` se comportan mejor en escenarios con lenguaje natural (`naturales`), mientras que modelos como `distiluse-base-multilingual` destacan cuando las consultas tienen estructura definida (`con_fecha`).

6.4 Conclusión y modelo elegido

Tras un análisis comparativo entre precisión, estabilidad y cobertura multilingüe, el modelo seleccionado para producción fue:

paraphrase-multilingual-mpnet-base-v2

Este modelo ofrece:

– Alto rendimiento en todas las métricas (ver NDCG@10, Recall@10).

– Buen comportamiento tanto en queries estructuradas como en lenguaje libre.
– Soporte multilingüe, esencial para escenarios reales en varias lenguas.

Table 2: Rendimiento comparativo completo de modelos de embeddings

Modelo	Tipo Query	NDCG@10	Precision@10	Recall@10
Familia MiniLM				
all-MiniLM-L12-v2	con_fecha	0.213	0.053	0.178
all-MiniLM-L12-v2	estándar	0.461	0.140	0.467
all-MiniLM-L12-v2	naturales	0.362	0.093	0.311
all-MiniLM-L6-v2	con_fecha	0.315	0.067	0.222
all-MiniLM-L6-v2	estándar	0.430	0.113	0.378
all-MiniLM-L6-v2	naturales	0.302	0.073	0.244
Familia MPNet				
all-mpnet-base-v2	con_fecha	0.327	0.073	0.244
all-mpnet-base-v2	estándar	0.388	0.107	0.356
all-mpnet-base-v2	naturales	0.290	0.067	0.222
Modelos Multilingües				
distiluse-base-multilingual-cased-v1	con_fecha	0.294	0.120	0.400
distiluse-base-multilingual-cased-v1	estándar	0.443	0.147	0.489
distiluse-base-multilingual-cased-v1	naturales	0.412	0.140	0.467
paraphrase-multilingual-mpnet-base-v2	con_fecha	0.412	0.107	0.356
paraphrase-multilingual-mpnet-base-v2	estándar	**0.507**	**0.153**	**0.511**
paraphrase-multilingual-mpnet-base-v2	naturales	0.500	0.147	0.489
sentence-transformers/LaBSE	con_fecha	0.324	0.113	0.378
sentence-transformers/LaBSE	estándar	0.324	0.100	0.333
sentence-transformers/LaBSE	naturales	0.278	0.100	0.333
Modelos MSMARCO				
sentence-transformers/msmarco-MiniLM-L6-cos-v5	con_fecha	0.285	0.080	0.267
sentence-transformers/msmarco-MiniLM-L6-cos-v5	estándar	0.425	0.140	0.467
sentence-transformers/msmarco-MiniLM-L6-cos-v5	naturales	0.203	0.087	0.289
sentence-transformers/msmarco-distilbert-base-v3	con_fecha	0.350	0.093	0.311
sentence-transformers/msmarco-distilbert-base-v3	estándar	0.503	0.120	0.400
sentence-transformers/msmarco-distilbert-base-v3	naturales	0.328	0.107	0.356

Nota: Valores en negrita indican los mejores resultados por métrica. Líneas grises claras separan subgrupos dentro de cada familia.

Conclusión: el análisis comparativo permitió seleccionar un modelo robusto y versátil que mejora la calidad de las búsquedas semánticas y la experiencia de usuario en el sistema.

Declaración autocrítica del proyecto

Este proyecto consistió en desarrollar un sistema inteligente que organiza eventos usando distintos "agentes" especializados, como si fueran personajes con tareas definidas. A continuación, se detallan los logros, las dificultades y algunas ideas de mejora.

Logros alcanzados

– **Arquitectura bien organizada:** el sistema tiene varios agentes independientes (scraping, procesamiento, grafo, búsqueda, optimización) que trabajan juntos de forma coordinada.
– **Recolección masiva de eventos:** usando las APIs de SeatGeek, Ticketmaster y PredictHQ se recolectaron más de 2000 eventos.
– **Procesamiento enriquecido:** los datos crudos fueron transformados a un formato limpio, completo y con campos útiles como título, artistas, lugar, etc.

- **Búsqueda semántica inteligente:** permite al usuario buscar eventos usando lenguaje natural, con filtros por ciudad, categoría o fechas.
- **Grafo de conocimiento:** se relacionan eventos con artistas, fechas, ciudades y categorías. También se incluye un análisis de similitud entre eventos.
- **Agenda recomendada:** el sistema genera una agenda automática optimizada usando heurísticas, evitando eventos solapados o muy lejos entre sí.
- **Fallback dinámico:** si no se encuentran suficientes eventos, se buscan nuevos desde una fuente externa y se integran automáticamente.
- **Interfaz visual amigable:** se creó una aplicación en Streamlit que permite buscar eventos, ver detalles y generar una agenda personalizada.

Dificultades e insuficiencias

- **Agenda poco configurable:** el número de eventos es fijo (5), y no se permite elegir horarios o restricciones de transporte.
- **Fallback limitado:** actualmente solo se usa una fuente por búsqueda, no se combinan resultados de varias.
- **Sin perfil del usuario:** no se guarda información como búsquedas anteriores, gustos o favoritos.
- **Falta soporte multilenguaje:** solo está en español e ingles , no hay opción para inglés u otros idiomas.
- **No hay base de datos : actualmente se trabaja con archivos en formato json**

Propuestas de mejora

- Dejar que el usuario indique cuántos eventos quiere y qué tipo de horarios prefiere.
- Combinar varias fuentes externas en el fallback en vez de usar solo una.
- Crear un archivo de perfil con las preferencias y eventos favoritos del usuario.
- Agregar opción para cambiar de idioma en la interfaz.

Esta reflexión final nos ayudó a entender qué aspectos del sistema ya funcionan bien y en cuáles se puede seguir trabajando para mejorarlo.

Bibliografía

1. **Conferencias de Sistemas de Recuperación de Información**, curso 2024–2025.
   Facultad de Matemática y Computación, Universidad de La Habana.
2. **Material de estudio y conferencias de Simulación**, curso 2024–2025.
   Facultad de Matemática y Computación, Universidad de La Habana.
3. **Búsqueda vectorial y embeddings**
   - ¿Qué es la búsqueda de vectores? *IBM*, 2024.
     `https://www.ibm.com/es-es/think/topics/vector-search`
4. **Evaluación de modelos: precisión, recall y métricas**
   - Ejemplo de métricas: Curvas de precisión y recall. *Video educativo*, HUBIQ Energía e Innovación Tecnológica, 2023.
     `https://www.youtube.com/watch?v=GXdBlhWnK0Y`