# TuristAI: Un Asistente Virtual para el Turismo

Dayan Cabrera Corvo[1], Eveliz Espinaco Milián[2], and Michell Viu Ramirez[3] *

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


* Repositorio del proyecto: `https://github.com/DayanCabrera2003/TuristAI`

## 1. Introducción

TuristAI es un asistente virtual diseñado para mejorar la experiencia turística en Cuba. Utilizando tecnologías de Inteligencia Artificial, Sistemas de Recuperación de Información y Simulación este sistema proporciona información personalizada sobre destinos turísticos, recomendaciones de actividades y asistencia en tiempo real. El objetivo principal es facilitar la planificación del viaje y enriquecer la experiencia del usuario mediante interacciones naturales y eficientes. El proyecto se enmarca dentro de un sistema multiagente donde cada componente cumple un rol específico, contribuyendo a una solución integral y desacoplada.

## 2. Descripción del Tema: El Turismo en Cuba

### 2.1. El Turismo como Pilar de la Economía Cubana:

El turismo en Cuba ha sido tradicionalmente catalogado como la "locomotora de la economía cubana"desde los años cincuenta, representando uno de los sectores de mayor crecimiento a escala mundial. Esta industria constituye una de las principales fuentes de divisas frescas para el país, permitiendo equilibrar la balanza comercial y financiar prioridades en sectores estratégicos como la salud, la educación y la infraestructura.

### 2.2. Situación Actual del Sector Turístico Cubano:

La industria turística cubana enfrenta actualmente una de las crisis más severas de su historia [1][2]. En 2024, Cuba recibió únicamente 2,2 millones de visitantes internacionales, representando una disminución del 9,6 % respecto al año anterior y constituyendo la peor cifra en 17 años, excluyendo los años de pandemia [3] [4]. Esta cifra contrasta dramáticamente con los objetivos iniciales del gobierno de alcanzar 3,2 millones de turistas, meta que posteriormente fue reducida a 2,7 millones.
El panorama se torna aún más preocupante al analizar las tendencias del primer trimestre de 2025, donde se registró una caída del 29,7 % en el número de visitantes respecto al mismo período del año anterior. Entre enero y abril de 2025, Cuba recibió 991,103 viajeros internacionales, lo que representa una disminución de 265,777 visitantes comparado con el mismo período de 2024. [3] [4]

### 2.3. Justificación del Proyecto: Asistente Turístico Digital para Cuba

Un asistente turístico digital para Cuba podría abordar múltiples desafíos identificados en el sector, incluyendo la fragmentación de información, la necesidad de orientación personalizada, y la optimización de la experiencia turística. El sistema propuesto podría integrar información sobre destinos, servicios, condiciones actuales del país, opciones de hospedaje, gastronomía, transporte, y actividades culturales en una plataforma única y accesible.

## 3. Solución Implementada

La solución propuesta se estructuró como un sistema multiagente, donde cada agente asumió responsabilidades específicas dentro del flujo general del sistema. Se diseñó una arquitectura modular y desacoplada, permitiendo la interacción coordinada entre agentes encargados de la recolección de datos, procesamiento de información, y optimización de itinerarios turísticos.
El flujo del sistema se inició con la recolección automatizada de información relevante sobre destinos, alojamientos y actividades turísticas mediante agentes

de crawling y scraping. Posteriormente, los datos fueron procesados y almacenados en una base de conocimiento estructurada, facilitando su recuperación eficiente. Un agente especializado en Recuperación de Información y Generación Aumentada (RAG) se encargó de responder consultas del usuario, integrando fragmentos relevantes de la base de datos y generando respuestas personalizadas mediante modelos de lenguaje natural. Finalmente, un planificador inteligente utilizó técnicas de optimización para construir itinerarios turísticos adaptados a las preferencias y restricciones del usuario, completando así el ciclo del sistema.

### 3.1. Componentes del Sistema

A continuación se describen con mayor detalle cada uno de los componentes que conforman la aplicación:

**Retrievel-Augmented Generation (RAG) [5]:** El objetivo principal del componente RAG desarrollado para este proyecto es mejorar la precisión, relevancia y fundamentación de las respuestas generadas por el asistente conversacional TuristAI. Para lograrlo, el componente RAG integra mecanismos de recuperación semántica de información desde una base de conocimiento local, combinados con la capacidad generativa de un modelo de lenguaje avanzado.
De manera específica, el componente RAG busca:

- Enriquecer las respuestas del chatbot con información actualizada y específica del dominio turístico cubano, proveniente de fuentes estructuradas y curadas localmente.
- Reducir la incidencia de respuestas genéricas, imprecisas o desinformadas por parte del modelo generativo, proporcionando contexto relevante extraído dinámicamente en función de la consulta del usuario.
- Optimizar la eficiencia y la experiencia de usuario, precalculando y almacenando los embeddings de los documentos para acelerar la recuperación y respuesta.

La solución desarrollada consiste en un sistema que combina recuperación semántica de información y generación de lenguaje natural para responder preguntas turísticas sobre Cuba.
El sistema integra dos componentes principales:

- Un recuperador semántico basado en embeddings generados con **sentence-transformers** (all-MiniLM-L6-v2), que permite buscar fragmentos relevantes en una base de conocimiento local estructurada en archivos JSON.
- Un modelo generativo (Gemini de Google) que, a partir de los fragmentos recuperados y la consulta del usuario, genera una respuesta en lenguaje natural.

El flujo de trabajo del sistema es el siguiente :

1. Los archivos JSON se procesan para extraer y concatenar los textos relevantes (título y fragmentos de secciones).

2. Cada texto se divide en chunks solapados para mejorar la granularidad de la recuperación.
3. Los embeddings de los chunks se precalculan y almacenan, optimizando el tiempo de respuesta.
4. Ante una consulta, se recuperan los fragmentos más relevantes y se construye un prompt enriquecido para el modelo generativo, en caso de que la información recuperada no sea relevante se usa un crawler dinámico que recupera información según la consulta del usuario, de esta forma el modelo de lenguaje siempre tendrá el contexto necesario para dar una respuesta adecuada.

**Consideraciones de implementación**

- Eficiencia: Se priorizó el precálculo y almacenamiento de embeddings para evitar cálculos redundantes y mejorar la experiencia de usuario.
- Modularidad: Toda la lógica de chunking, embeddings y recuperación se encapsuló en la clase ChatUtils, facilitando la mantenibilidad y futuras mejoras.
- Flexibilidad: El sistema permite ajustar el tamaño de los chunks, el solapamiento y el número de fragmentos recuperados, adaptándose a diferentes necesidades y volúmenes de datos.
- Usabilidad: Se diseñó el prompt para que el modelo generativo use el contexto recuperado, pero también pueda responder con su conocimiento general, evitando respuestas evasivas.

**Crawlers [8] Objetivos del componente**

- Automatizar la recolección de información turística desde diversas fuentes en línea.
- Soportar tanto sitios estáticos como dinámicos mediante técnicas adaptativas.
- Garantizar eficiencia, concurrencia controlada y almacenamiento estructurado.
- Permitir la integración fluida con el resto de la arquitectura multiagente.

**Tecnologías utilizadas**

El desarrollo se realizó en Python e integró librerías especializadas:

- `requests`: Descarga eficiente de contenido HTML en sitios estáticos.
- `Selenium`: Automatización de navegación y scraping en sitios dinámicos (JavaScript).
- `BeautifulSoup`: Extracción y procesamiento de información estructurada.
- `fake_useragent`: Rotación de agentes de usuario para evitar bloqueos.
- `threading`: Ejecución multihilo y sincronización.
- `duckduckgo-search`: Búsqueda web rápida para crawling dinámico.
- `os`, `json`, `hashlib`: Manejo de archivos y persistencia.

### Arquitectura y Funcionamiento del Crawler Multihilo (`crawler2.py`)

#### Estructura general y flujo de ejecución

El archivo `crawler2.py` implementa un **crawler multihilo** capaz de:

- Realizar crawling tradicional (requests/Selenium) y dinámico (requests + búsqueda web).
- Extraer y almacenar información estructurada (título, fragmentos, teléfonos, emails) en formato JSON.
- Organizar los datos por dominio y tipo de fuente (`data/`, `data_dynamic/`).
- Limitar la profundidad y la cantidad de descargas por semilla para evitar sobrecarga.
- Sincronizar el acceso a recursos compartidos entre hilos.

#### Componentes principales y métodos clave

- **Clase** `CrawlerState`: Gestiona el estado compartido entre hilos, incluyendo URLs visitadas, límites por semilla, asignación de dominios y sincronización mediante `threading.Lock`.
- **Función** `process_page`: Descarga y procesa una página web. Usa `requests` para sitios estáticos y Selenium para dinámicos, extrae información relevante y guarda los resultados en la carpeta correspondiente dentro de `agents`.
- **Función** `parallel_crawler`: Orquesta la ejecución multihilo. Cada hilo ejecuta un `worker` que toma URLs de una cola sincronizada, procesa la página y añade nuevos enlaces si corresponde.
- **Función** `run_crawler`: Punto de entrada principal. Permite crawling por semillas o por consulta dinámica (usando `duckduckgo-search`), ajustando la profundidad y el modo de operación.
- **Módulos auxiliares**: `extractor.py` (extracción de enlaces y datos), `utils.py` (manejo de archivos), `config.py` (configuración global).

### Scrapper Especializado para CubaTravel (`scrapper.py`)

#### Propósito y enfoque

El archivo `Scrapper.py` implementa un agente especializado para la extracción de información desde el portal `cubatravel.cu`, que presenta una interfaz altamente dinámica y dependiente de JavaScript. Este scrapper:

- Navega automáticamente por todas las pestañas (Hoteles, Casas, Tours, etc.) y destinos disponibles.
- Interactúa con menús desplegables, botones y formularios usando Selenium.
- Extrae tarjetas de resultados y accede a los detalles de cada resultado para enriquecer la información.
- Guarda los resultados en archivos JSON y HTML de depuración para análisis posterior.

#### Funcionamiento y métodos principales

- **Función** `scrape`: Orquesta todo el proceso de scraping, iterando por pestañas y destinos, seleccionando opciones, lanzando búsquedas y extrayendo resultados.
- **Función** `create_driver`: Configura Selenium en modo headless para evitar la apertura de ventanas y mejorar el rendimiento.
- **Función** `click_tab` y `abrir_selector_destinos`: Automatizan la navegación por la interfaz, asegurando que los elementos estén presentes y visibles antes de interactuar.
- **Función** `extract_result_cards` y `extract_card_data`: Extraen la información relevante de cada tarjeta de resultado, incluyendo nombre, descripción, dirección, tarifa, precio y estrellas.
- **Función** `save_debug_html`: Guarda el HTML de la página en cada paso para facilitar la depuración y el análisis manual.

**Manejo de errores y robustez**

- El scrapper implementa múltiples mecanismos de espera y reintento para manejar la carga dinámica y posibles fallos de la interfaz.
- Se guarda el HTML de depuración en caso de error para facilitar el troubleshooting.
- Se utiliza logging detallado para registrar el progreso y los problemas encontrados.

**Justificación de Decisiones de Diseño e Implementación**

El diseño de ambos agentes se fundamentó en principios de eficiencia, escalabilidad y robustez, alineados con las mejores prácticas en el desarrollo de sistemas de scraping y arquitecturas multiagente. A continuación, se detallan y justifican las principales decisiones tomadas:

**Ejecución Multihilo y Gestión del Estado Compartido**
Se optó por una arquitectura multihilo utilizando el módulo `threading` de Python para maximizar el aprovechamiento de los recursos del sistema y acelerar la recolección de datos. El uso de `Lock` garantiza la consistencia del estado global.

**Soporte a Sitios Estáticos y Dinámicos**
El crawler distingue entre sitios estáticos (requests) y dinámicos (Selenium), permitiendo una cobertura completa y eficiente de portales turísticos modernos.

**Normalización y Filtrado de Enlaces**
La normalización de URLs y el filtrado por dominios válidos aseguran que el crawler no se desvíe hacia sitios irrelevantes, manteniendo el foco en el dominio turístico cubano.

**Límites por Semilla y Profundidad de Rastreo**
Se establecieron límites estrictos de descargas por semilla y una profundidad máxima de rastreo para evitar la sobrecarga de servidores y cumplir con buenas prácticas de scraping responsable.

**Estructura Modular y Persistencia**
El código se organizó en módulos independientes (`utils.py`, `extractor.py`,

etc.), facilitando el mantenimiento y la extensibilidad. La persistencia de los datos en formato JSON permite una integración sencilla con etapas posteriores.

**Gestión de Contenido Dinámico**
Para sitios con carga dinámica, se implementaron rutinas específicas de interacción (scroll, clic en botones de "ver más"), justificadas por la necesidad de acceder a información que no está disponible en el HTML inicial.

**Registro y Manejo de Errores**
Se incorporó un sistema de logging y guardado de HTML de depuración para facilitar la depuración y el monitoreo del proceso.

**Consideraciones de Implementación**
Durante el desarrollo se tuvieron en cuenta las siguientes consideraciones:

- Se evitó la sobrecarga de servidores estableciendo un límite de descarga por semilla.
- Se aplicó normalización de enlaces y filtrado por dominios válidos.
- Se priorizó el desacoplamiento funcional dividiendo la lógica en módulos independientes.
- Se definió una profundidad máxima de rastreo para controlar la expansión exponencial.
- Se garantizó que todas las carpetas de datos se creen dentro de la carpeta `agents`, independientemente del modo de ejecución.

**Resultados de los Componentes de Crawling**
El crawler multihilo fue ejecutado con un conjunto inicial de URLs semilla, representativas del dominio turístico cubano. Se logró recolectar y almacenar más de 2000 documentos en formato JSON, organizados por dominio. El scrapper especializado permitió extraer información detallada y enriquecida de `cubatravel.cu`, cubriendo todas las pestañas y destinos disponibles. Ambos sistemas demostraron eficiencia, cobertura y robustez, manteniendo el control de concurrencia y el cumplimiento de los límites establecidos.

### 3.2. Planificador Inteligente

El componente de planificación genera itinerarios turísticos personalizados mediante un enfoque híbrido que combina:

- **Recuperación de información**: Uso de RAG (Retrieval-Augmented Generation) para consultar actividades relevantes
- **Modelado CSP**: Formalización como problema de satisfacción de restricciones (presupuesto, tiempo, preferencias)
- **Optimización metaheurística**: Búsqueda de soluciones cuasi-óptimas en espacios combinatorios complejos

#### Flujo de trabajo

1. **Fase de recuperación**:

```
query = "Museos, Playas en La Habana, Varadero"
resultado = rag.ask(query, schema_json)
actividades = json.loads(extraer_json(resultado))
```

2. **Modelado del problema**:
   - *Variables*: Actividades turísticas seleccionables
   - *Dominio*: $D = \{\text{actividad}_1, \text{actividad}_2, \ldots, \text{actividad}_n\}$
   - *Restricciones*: $\sum \text{costos} \leq \text{presupuesto}$
     $|\text{actividades}| \leq \text{días disponibles})$
     $\text{preferencias} \subseteq \text{tipos de actividades}$
3. **Optimización**:

```
# Configuración según objetivos del usuario
optimizador = MetaheuristicasItinerario(
    min_presupuesto=True,
    max_lugares=False
)

# Ejecución de algoritmo genético
itinerario_optimo = optimizador.algoritmo_genetico_itinerario(
    actividades_disponibles,
    generaciones=50
)
```

**Implementación de metaheurísticas Algoritmo Genético (GA)**:

- *Representación*: Vector de actividades
- *Operadores*:

```
def cruzar(padre1, padre2):
    punto = random.randint(1, len(padre1)-1)
    return padre1[:punto] + padre2[punto:]

def mutar(itinerario, actividades, prob=0.1):
    for i in range(len(itinerario)):
        if random.random() < prob:
            itinerario[i] = random.choice(actividades)
```

- *Selección*: Elitismo (mejores individuos conservados)

**PSO (Particle Swarm Optimization)**:

- *Partículas*: Soluciones candidatas (combinaciones de actividades)
- *Actualización de posición*: $x_i(t+1) = x_i(t) + v_i(t+1)$
- *Parámetros*: Inercia ($\omega = 0{,}5$), factores cognitivo/social ($c_1 = c_2 = 1{,}5$)

**Función de evaluación** La calidad de soluciones se calcula mediante:

```
def evaluar_itinerario(actividades):
    evaluacion = 0
    # Recompensa por preferencias cubiertas
    for pref in preferencias:
        if pref in actividad: evaluacion += 2

    # Penalización por exceso presupuestario
    if costo_total > presupuesto:
        evaluacion -= (costo_total - presupuesto)/10

    # Penalización por preferencias no cubiertas
    evaluacion -= len(preferencias_no_cubiertas)
    return evaluacion
```

**Ventajas del enfoque**

- **Escalabilidad**: Maneja $O(n!)$ combinaciones con complejidad $O(k \cdot n^2)$
- **Flexibilidad**: Adaptación a distintos objetivos mediante flags:

```
min_presupuesto=True   # Modo económico
max_lugares=True       # Modo maximalista
```

- **Robustez**: Soluciones factibles bajo restricciones complejas

### 3.3. Experimentación

Para determinar la metaheurística óptima para el sistema de planificación turística, se diseñó un protocolo experimental riguroso que incluyó:

**Diseño experimental**

- **Muestra**: 100 simulaciones de usuarios con preferencias generadas aleatoriamente
- **Parámetros variados**:
  - Presupuesto: \$200-\$1000 (distribución uniforme)
  - Días disponibles: 3-10 días
  - Preferencias temáticas: 3-8 categorías por usuario
  - Localizaciones: 1-5 provincias cubanas
- **Metaheurísticas evaluadas**:
  1. Algoritmo Genético (GA)
  2. Enjambre de Partículas (PSO)
  3. Variante híbrida GA-PSO

- **Métrica de evaluación**: Función de evaluación unificada:

$$\text{Score} = \underbrace{2 \times P_C}_{\text{Preferencias cubiertas}} \underbrace{- 0{,}1 \times \max(0, C_T - P)}_{\text{Exceso presupuesto}} - \underbrace{0{,}5 \times P_N}_{\text{Preferencias no cubiertas}} \tag{1}$$

donde:
- $P_C$: Preferencias cubiertas
- $C_T$: Costo total
- $P$: Presupuesto
- $P_N$: Preferencias no cubiertas

### 3.4. Interfaz de la Aplicación

La interfaz de la aplicación se implementó utilizando la biblioteca Streamlit, permitiendo una interacción sencilla e intuitiva con el usuario. A través de esta interfaz, el usuario pudo introducir sus preferencias, restricciones y consultas de manera natural. El sistema presentó los resultados de forma clara y estructurada, mostrando recomendaciones, itinerarios sugeridos y detalles relevantes de cada destino o actividad. Además, la interfaz facilitó la retroalimentación del usuario, permitiendo ajustar las recomendaciones en función de nuevas necesidades o intereses, y garantizando una experiencia personalizada y dinámica durante todo el proceso de planificación turística.

## 4. Declaración Autocrítica y Propuestas de Mejora

Entre los principales logros de la implementación se destaca la integración efectiva de técnicas de Recuperación de Información, generación aumentada y planificación inteligente en un sistema multiagente modular. El uso de un *crawler* dinámico permitió mitigar la falta de información local en tiempo real, mejorando la cobertura y actualidad de las respuestas. No obstante, se identificaron insuficiencias, principalmente relacionadas con la dependencia de la calidad y cantidad de los datos almacenados localmente. En varios casos, la información recuperada no fue lo suficientemente relevante, lo que afectó la precisión de las recomendaciones iniciales. Además, la invocación del *crawler* dinámico, aunque efectiva, incrementó el tiempo de respuesta en ciertas situaciones.
Como propuestas de mejora, se sugiere la aplicación de técnicas avanzadas en el componente RAG, tales como el uso de modelos de recuperación más sofisticados, la expansión semántica de consultas y la integración de fuentes de datos adicionales. Asimismo, se recomienda optimizar el proceso de crawling dinámico mediante la priorización de enlaces y la actualización periódica de la base de conocimiento local, con el objetivo de reducir la necesidad de búsquedas en tiempo real y mejorar la relevancia de la información recuperada.

## Referencias


1. *El turismo en Cuba se apaga: caída récord en ingresos y visitantes*. CubaSiglo21. 2025.
   URL: https://cubasiglo21.com/el-turismo-en-cuba-se-apaga-caida-record-en-ingresos-y-visitantes/
2. *Desafíos y Declive: La Crisis Permanente de la Industria Turística Cubana*. CubaSiglo21. 2025.
   URL: https://cubasiglo21.com/desafios-y-declive-la-crisis-permanente-de-la-industria-turistica-cub
3. *Cuba recibió 2,2 millones de turistas en 2024, la peor tasa en 17 años sin contar la covid*. Swissinfo.ch. 2025.
   URL: https://www.swissinfo.ch/spa/cuba-recibi%C3%B3-2,2-millones-de-turistas-en-2024,-la-peor-tasa-en-17-a%C3%B1os-sin-contar-la-covid/88793383
4. *Turismo en Cuba cerró 2024 con 2.2 millones de visitantes internacionales*. ExcelenciasCuba. 2025.
   URL: https://www.excelenciascuba.com/turismo/turismo-en-cuba-cerro-2024-con-22-millones-de-visitantes-internacionales
5. Zhao, Penghao; Zhang, Hailin; Yu, Qinhan; Wang, Zhengren; Geng, Yunteng; Fu, Fangcheng; Yang, Ling; Zhang, Wentao; Jiang, Jie; Cui, Bin. *Retrieval-augmented generation for AI-generated content: A survey*. 2024. arXiv preprint arXiv:2402.19473.
6. Oliva-Santos, R. (2008). Socialización de la información y el conocimiento en la Web. *Revista Ciencias Matemáticas*, 24, 93-103.
7. Baeza-Yates, R., Ribeiro-Neto, B. (2002). *Modern Information Retrieval*. pp. 367-395.
8. Manning, C. D., Raghavan, P., Schütze, H. (2007). *An Introduction to Information Retrieval*. pp. 399-436.