CarrenoGo - La Ventana al Orinoco

🛡️ Serverless Lexical RAG Chatbot

📚 Documentación

📖 DESCRIPCIÓN

CarrenoGo es un ecosistema digital de turismo sostenible para el departamento del Vichada, Colombia. Conecta directamente a turistas nacionales e internacionales con operadores locales certificados de Puerto Carreño, eliminando el 30% de comisión que cobraban las agencias intermediarias.

Arquitectura Técnica

El sistema utiliza una **Artificial Narrow Intelligence (ANI)** basada en una arquitectura de **Retrieval-Augmented Generation (RAG)** con **recuperación léxica** (algoritmo BM25), lo que permite al agente responder consultas turísticas con información precisa y minimizar alucinaciones. En cada consulta, el sistema recupera dinámicamente los fragmentos más relevantes de la **base de conocimiento** (Local Markdown-Based Knowledge Corpus) y los inyecta como contexto al modelo de lenguaje vía API externa.

Esta elección técnica tiene como objetivo **privilegiar estructuras legibles por humanos y máquinas**, garantizando que la base de conocimiento sea tan fácil de auditar y editar por un administrador como de procesar por el modelo de lenguaje.

¿Qué problema resuelve?

Los operadores turísticos del Vichada tienen una oferta de clase mundial — safaris en la sabana llanera, expediciones fluviales por el Orinoco, inmersiones culturales con comunidades Sikuani, pesca del payara — pero son prácticamente invisibles para el mercado global, tienen poca o nula presencia digital, dependen de agencias que se quedan con buena parte del valor generado, y no cuentan con herramientas para gestionar su oferta de forma autónoma.

CarrenoGo democratiza el acceso a la inteligencia artificial para que esos operadores puedan competir en igualdad de condiciones: un agente IA disponible 24/7 atiende las consultas de los turistas, el precio va directo a la comunidad local, y el catálogo se actualiza con un simple `git push`.

Stack tecnológico

Componente	Tecnología
Server & Logic	Serverless Edge (Cloudflare Workers)
User Interface	HTML + CSS + JS — PWA instalable

Componente	Tecnología
AI Agent	Google Gemini 2.5 Flash
Retrieval Engine	Lexical RAG / BM25 (JS Implementation)
Base de Conocimiento	Local Markdown-Based Knowledge Corpus
Notificaciones	Resend (email) + Google Sheets (CRM)
CI/CD	GitHub → Serverless Edge Integration

🚀 DESPLIEGUE

HTML + Chatbot

Crear la interfaz web con el chat integrado. El chatbot hace `fetch` a `/api/chat` (ruta relativa, nunca directamente a la API del proveedor). Todo el HTML vive dentro del Worker como una constante:

```
const HTML = `<!DOCTYPE html>...`;
```

Worker

El `worker.js` contiene tres partes:

1. `const HTML` — toda la interfaz (HTML + CSS + JS del chat)
2. `const SYSTEM_PROMPT` — personalidad e instrucciones del agente IA
3. **Bloque del proveedor de IA** — claramente delimitado con comentarios `═══` para fácil modificación

API Key de IA

Proveedor	Dónde obtenerla
Google - Gemini	aistudio.google.com → Get API key
Anthropic - Claude	console.anthropic.com → API Keys → Create Key
OpenAI - ChatGPT	platform.openai.com → API keys → Create new
DeepSeek	platform.deepseek.com → API Keys → Create new

Git

Para gestionar el código y permitir la integración continua con el proveedor de la plataforma:

- Crear el repositorio: Crea un nuevo repositorio en tu cuenta de GitHub (ej. proyecto-xxx).
- Subir archivos: En tu terminal local, inicializa el repositorio y sube los siguientes archivos esenciales:

Contenido

- `worker.js`: Contiene el Worker junto con la estructura HTML que se servirá.

- `data.md`: Contiene los datos en formato MD (Local Markdown-Based Knowledge Corpus).

> El repositorio contiene otros archivos y carpetas, pero son para soporte técnico

Serverless Edge

La plataforma elegida, de tipo Serverless Edge, para el despliegue del prototipo es: Cloudflare

- Ir a **Cloudflare → Workers & Pages → Create**
- Elegir el nombre de la URL
- Seleccionar **"Empiece con ¡Hola mundo!"**
- Click en **Deploy**

Variables

En el Worker recién creado:

1. API IA

- **Settings → Variables and Secrets → Add**
- Nombre: `API_KEY` (exactamente)
- Valor: la clave del proveedor
- Marcar como **Secret** (ícono de candado)
- Click en **Save**

1. API Resend

- **Settings → Variables and Secrets → Add**
- Nombre: `RESEND_KEY` (exactamente)
- Valor: la clave del proveedor
- Marcar como **Secret** (ícono de candado)
- Click en **Save**

> Los datos de los clientes se enviarán a una tabla de Google Sheets al correo del SysAdmin/Dev del proyecto, por tanto asegúrese de protegerlos.

URL

Abrir la URL del Worker en el navegador. La página debe cargar correctamente. Bajar al chatbot, escribir `hola` y confirmar que el agente responde; luego preguntar por una oferta específica incluida en Local Markdown-Based Knowledge Corpus.

🤖 PROVEEDORES API IA

En el `worker.js` hay un bloque claramente marcado:

```
// ═══════════════════════════════════════════════════════════════
// BLOQUE DE PROVEEDOR DE IA — edita aquí para cambiar de modelo
// Variable de entorno requerida: API_KEY
//
```

```
// OPCIÓN ACTIVA: Google Gemini
// ──────────────────────────────────────────────────────────────────────
  ... código Gemini activo ...

// OPCIÓN ALTERNATIVA: Anthropic Claude
// Para activar: comenta el bloque Gemini y descomenta esto
// ──────────────────────────────────────────────────────────────────────
// ... código Claude comentado ...
// ══════════════════════════════════════════════════════════════════════
```

Para cambiar:

1. Abrir el Worker → **Edit code**
2. Comentar el bloque activo (agregar `//` al inicio de cada línea)
3. Descomentar el bloque alternativo (quitar `//` del inicio de cada línea)
4. Actualizar el valor de `API_KEY` en Settings con la clave del nuevo proveedor
5. **Save & Deploy**

💬 INTERACCIÓN

Clientes

El visitante completa el formulario de la sección **#contacto** con los siguientes datos: nombre completo, teléfono, correo electrónico, experiencia de interés y mensaje. Al pulsar **Enviar**, la información viaja cifrada por HTTPS hacia el Worker y se procesa en dos destinos simultáneos:

1. **Correo electrónico del proyecto** — vía Resend, con todos los datos formateados para revisión inmediata.
2. **Hoja de Google Sheets** — vía Apps Script Webhook, donde queda registrada la solicitud para seguimiento del SysAdmin/Dev.

> El SysAdmin/Dev recibe la notificación, revisa ambos registros y contacta al cliente directamente para coordinar la reserva con el operador local. Los datos personales se transmiten sobre HTTPS y no son accesibles públicamente. El tratamiento de esta información debe realizarse en cumplimiento de la Ley de Protección de Datos Personales y las políticas de privacidad que el proyecto defina para su fase de producción.

Operadores

Primera vez: el operador envía su oferta detallada al correo electrónico del proyecto. El SysAdmin/Dev la estructura en `data.md` siguiendo el formato del catálogo y hace `git push` para que quede disponible en el sistema de inmediato.

Acceso al portal: una vez registrado, el operador recibe un enlace al aplicativo web de gestión. El acceso se realiza seleccionando su nombre en el listado de operadores e ingresando su correo electrónico. Desde allí puede visualizar su oferta tal como aparece en el catálogo, proponer modificaciones de precios, disponibilidad o descripción, y marcar una experiencia como no disponible temporalmente.

> Los cambios propuestos por el operador no se publican de forma automática. El SysAdmin/Dev recibe la solicitud, valida que la información sea correcta y coherente con el catálogo, y aplica la actualización en `data.md` mediante `git push`. Esto garantiza que el catálogo publicado siempre haya pasado por una revisión humana antes de estar disponible para los turistas.

Reseñas

- Una vez finalizada la experiencia, el SysAdmin/Dev envía al cliente un formulario de reseña con los siguientes campos: nombre, ciudad, fecha, experiencia realizada, calificación (1–5 estrellas) y comentario libre.
- El comentario está limitado a **280 caracteres** — suficiente para expresar una opinión clara y concisa, y compatible con el espacio disponible en la tarjeta de reseña de la landing page.
- Las reseñas recibidas no se publican de forma automática. El SysAdmin/Dev revisa el contenido, valida que sea pertinente y respetuoso, y agrega la reseña aprobada a la sección `## RESEÑAS` de `data.md` siguiendo el formato establecido:

```
### Nombre del cliente

**Ciudad:** Ciudad, País
**Fecha:** mes año
**Experiencia:** Nombre de la experiencia
**Estrellas:** 5
Texto del comentario (máximo 280 caracteres).
```

> Un `git push` posterior publica la reseña de forma inmediata en la landing page, sin necesidad de redesplegar el Worker.

🧠 ARQUITECTURA RAG

¿Qué es RAG?

RAG (Retrieval-Augmented Generation) es una arquitectura para sistemas de IA que combina dos capacidades: la búsqueda de información relevante en una base de conocimiento externa, y la generación de respuestas en lenguaje natural por parte de un LLM. La idea central es que el modelo no depende únicamente de lo que aprendió durante su entrenamiento, sino que puede consultar información actualizada y específica en el momento de responder. Esto reduce las alucinaciones y permite mantener el conocimiento actualizado sin necesidad de reentrenar el modelo.

Variantes RAG

> La siguiente clasificación abarca desde enfoques simples, como el Lexical RAG (BM25) y el RAG vectorial básico, centrados en la recuperación de información, hasta arquitecturas más complejas. Incluye variantes como Advanced y Modular RAG, que optimizan el flujo, Graph RAG para modelar relaciones, y enfoques como Agentic, Self-RAG y Corrective RAG, que incorporan mayor autonomía en el proceso de generación de respuestas.

Tipo	Descripción breve	Cuándo usarlo	Complejidad	Limitaciones
Lexical RAG / BM25 *(este proyecto)*	Búsqueda por coincidencia de términos sin embeddings. Recupera las secciones del documento con mayor puntaje BM25 antes de construir el prompt.	Primer paso hacia RAG real: agrega retrieval léxico sin infraestructura externa.	Baja — implementado en `worker.js` sin servicios adicionales.	No entiende semántica (solo coincidencia de términos); falla con sinónimos o queries ambiguas; depende mucho de cómo esté redactado el texto; baja precisión en preguntas complejas.
RAG Vectorial (naive)	Chunking → embeddings → búsqueda por similitud vectorial → chunks relevantes al prompt. Pipeline lineal básico y punto de referencia del campo.	Corpus mediano, caso de uso estándar.	Media — requiere modelo de embeddings y vector DB.	Sensible a la calidad del embedding; puede recuperar chunks irrelevantes; no optimiza contexto (ruido en el prompt); no maneja bien queries complejas o multi-hop; sin control de relevancia más allá de similitud.
Advanced RAG	Igual que naive pero con mejoras antes del retrieval (reescritura de consulta, expansión) y después (re-ranking, compresión de contexto). Pipeline base similar, mejor calidad.	Cuando el naive da resultados imprecisos.	Media-alta — múltiples componentes adicionales.	Mayor complejidad operativa; tuning difícil (muchos componentes); latencia más alta; riesgo de sobre-ingeniería; mejoras no siempre proporcionales al esfuerzo.

Tipo	Descripción breve	Cuándo usarlo	Complejidad	Limitaciones
Modular RAG	Cada componente (retriever, reranker, generador, memoria) es intercambiable. Más una arquitectura de diseño que un tipo específico. Frameworks como LangChain y LlamaIndex siguen este modelo.	Sistemas que requieren personalización por componente.	Alta — diseño modular desde el inicio.	Incrementa mucho la complejidad del sistema; integración y mantenimiento costosos; riesgo de inconsistencias entre componentes; requiere decisiones de diseño más sofisticadas desde el inicio.
Graph RAG	Construye un grafo de conocimiento donde los nodos son entidades y las aristas son relaciones. Recupera subgrafos relevantes en lugar de fragmentos de texto. Publicado por Microsoft en 2024.	Preguntas que requieren razonar sobre relaciones entre conceptos.	Alta — construcción y mantenimiento del grafo.	Alto costo de construcción y mantenimiento del grafo; requiere extracción de entidades y relaciones confiable; no escala bien con datos no estructurados o cambiantes; implementación compleja.
Agentic RAG	El LLM decide cuándo buscar, qué buscar y si los resultados son suficientes. Puede encadenar múltiples búsquedas. No es un pipeline lineal sino un bucle de razonamiento.	Preguntas complejas con razonamiento en múltiples pasos.	Muy alta — lógica de agente y múltiples llamadas.	Coste y latencia elevados (múltiples llamadas al LLM); comportamiento menos predecible; difícil de depurar; riesgo de loops innecesarios o decisiones subóptimas.

Tipo	Descripción breve	Cuándo usarlo	Complejidad	Limitaciones
Self-RAG	El modelo decide por sí mismo si necesita recuperar información o puede responder sin ella, y evalúa la calidad de los chunks recuperados. Requiere fine-tuning específico.	Minimizar llamadas innecesarias al retriever.	Muy alta — requiere fine-tuning del modelo base.	Requiere fine-tuning especializado; difícil de implementar y evaluar; depende fuertemente del modelo base; poco accesible en entornos sin control del modelo.
Corrective RAG (CRAG)	Si los chunks recuperados tienen baja relevancia, los descarta y hace una búsqueda web como respaldo antes de generar la respuesta.	Cuando la base de conocimiento local puede ser incompleta.	Alta — validación de relevancia + integración búsqueda web.	Dependencia de fuentes externas (web); posible inconsistencia entre fuentes; aumenta latencia; requiere mecanismos robustos de evaluación de relevancia; complejidad adicional en el fallback.

> La tendencia actual en el diseño de sistemas RAG apunta hacia enfoques como Agentic RAG y Graph RAG para la resolución de tareas complejas, mientras que Advanced RAG continúa siendo la estrategia predominante en aplicaciones de producción debido a su equilibrio entre rendimiento y complejidad. No obstante, en contextos reales, muchas herramientas —como NotebookLM, Implicit o Claude Projects— no se ajustan a una única categoría, sino que integran múltiples enfoques. En particular, suelen combinar técnicas propias de Advanced RAG con arquitecturas modulares y, en ciertos casos, incorporar comportamientos parcialmente agentic, configurando así sistemas híbridos a nivel de producto.

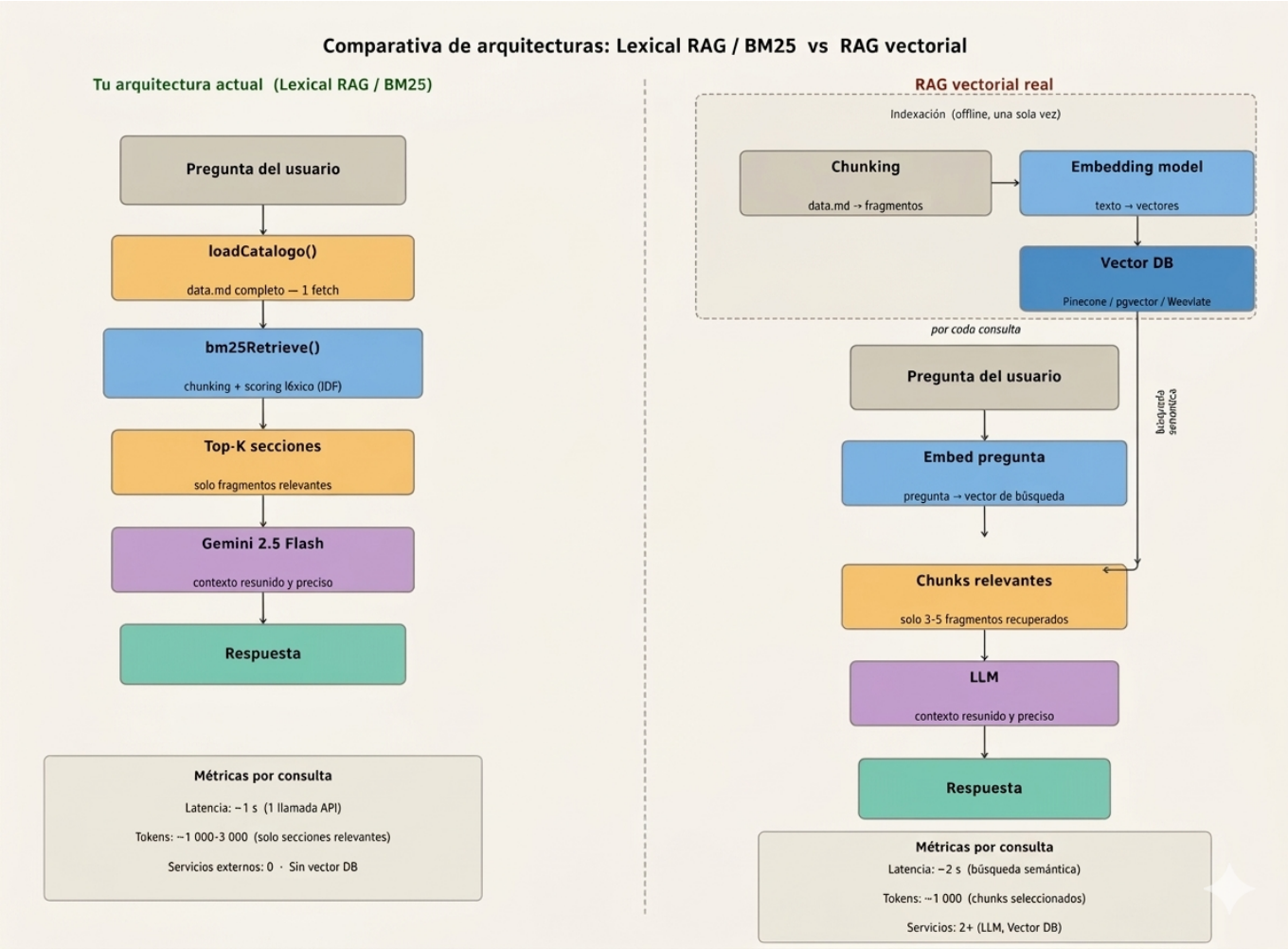
Elección de Arquitectura

Este proyecto implementa **Lexical RAG**, donde cada consulta pasa por un paso de recuperación léxica antes de construir el prompt: el corpus en Markdown se divide en secciones, se puntúa con **BM25** según la coincidencia con los términos de la pregunta, y solo las secciones con mayor puntaje se inyectan en el contexto del LLM.

La alternativa —RAG vectorial— convierte fragmentos en embeddings numéricos y los recupera por similitud semántica. Este enfoque mejora la comprensión de consultas indirectas, pero introduce

complejidad adicional (modelo de embeddings, vector DB y pipeline de re-indexación) que excede los requisitos de la Fase I.

Dimensión	Lexical RAG / BM25	RAG vectorial
Infraestructura	Cloudflare Worker + archivo estático	Worker + embedding API + vector DB
Complejidad	Baja — un solo `worker.js`	Alta — indexación, sincronización y gestión de chunks
Latencia	~1 s (BM25 local + 1 llamada al LLM)	~2 s (embedding + DB + LLM)
Costo operativo	$0	$20–70/mes
Actualización del catálogo	Editar `data.md` y hacer `git push`	Re-indexar fragmentos
Precisión (términos explícitos)	Alta — matching directo sobre nombres, precios y actividades	Alta
Precisión (consultas indirectas)	Media — depende del wording	Alta — comprensión semántica
Control y trazabilidad	Alto — secciones auditables por request	Medio — retrieval menos transparente



Lexical RAG es la elección adecuada para la Fase I por lo siguiente:

#	Razón	Detalle
1	**El retrieval léxico es suficiente**	Las consultas incluyen términos explícitos (experiencias, operadores, precios) que BM25 recupera con alta precisión.
2	**El corpus es estructurado**	Las secciones claras hacen que el matching sea predecible, auditable y fácil de depurar.
3	**Arquitectura sin dependencias externas**	Todo el sistema vive en un único Worker, sin necesidad de embeddings ni bases vectoriales.
4	**Actualización inmediata**	Un cambio en `data.md` se refleja directamente sin procesos de re-indexación.

> El principal límite aparece en consultas semánticas o composicionales (p. ej., "planes de 3 días" u "opciones para familias"), donde el retrieval puede ser subóptimo. Si el proyecto escala hacia catálogos más grandes o consultas más complejas, evolucionar hacia un enfoque híbrido sería el siguiente paso natural.

🔁 FASES DEL PIPELINE - LEXICAL RAG (BM25)

R — Recuperar (Retrieval)

Se mantiene un archivo `data.md` en el repositorio con la oferta turística clasificada (experiencias, operadores, precios, FAQ, etc.). Cloudflare sirve ese archivo como **asset estático** gracias al binding `ASSETS` declarado en `wrangler.toml`. En cada request al chatbot, el Worker recupera el texto completo, lo divide en secciones y aplica el algoritmo BM25 para seleccionar las piezas más relevantes según la consulta léxica del usuario:

```
const assetResponse = await env.ASSETS.fetch("http://placeholder/data.md");
const catalogoCompleto = await assetResponse.text();
const seccionesRelevantes = bm25Retrieve(catalogoCompleto, userQuery, topK = 5);
```

A — Aumentar (Augmentation)

El Worker construye un prompt aumentado antes de llamar a la API, reemplazando el marcador `{{CATALOGO}}` del `SYSTEM_PROMPT_BASE` únicamente con las secciones recuperadas por `BM25`. Esto evita procesar el documento completo, optimizando la ventana de contexto:

```
const SYSTEM_PROMPT = SYSTEM_PROMPT_BASE.replace("{{CATALOGO}}",
seccionesRelevantes);
```

El prompt final une tres piezas clave: la instrucción de sistema (personalidad), el contexto relevante del catálogo y el historial de la conversación.

G — Generar (Generation)

Este paquete se envía a la API del LLM. El modelo genera una respuesta basada estrictamente en las secciones del repositorio que BM25 identificó, lo que reduce drásticamente el consumo de tokens y minimiza las alucinaciones. Ventaja operativa: Para actualizar el catálogo basta con editar `data.md` y realizar un commit; el cambio se refleja de forma inmediata en la siguiente consulta sin necesidad de redesplegar el código del Worker.

¿Por qué este diseño es eficiente para el proyecto?

- Cero base de datos: No necesitas gestionar una base de datos vectorial (Vector DB).
- Mantenimiento simple: Todo vive en el repositorio de Git.
- Velocidad: Cloudflare Assets entrega el archivo `data.md` con latencia mínima.

⚠️ IMPORTANTE

- La `API_KEY` **nunca va en el HTML** — cualquiera la vería en el código fuente del navegador.
- Cloudflare Workers tiene plan gratuito con **100 000 requests/día** — suficiente para una demo académica.
- Si el chatbot responde "error de conexión": verificar que `API_KEY` esté guardada correctamente y que el Worker esté desplegado.
- Los datos del catálogo de experiencias son **referenciales** — los precios y disponibilidad deben verificarse directamente con cada operador.

🙏 IMÁGENES

Las imágenes del Hero son cortesía de la **Fundación Omacha**.
🔗 omacha.org/galeria-de-imagenes

Proyecto Académico UNAD - Fase I - Technology Readiness Level TRL 4