**Informe Evaluación Parcial 2: Desarrollo de un Agente Funcional**

* **Asignatura:** ISY0101 - Optativo Ingeniería de Soluciones con IA
* **Estudiantes:** Bryan Piña, Juan Castro
* **Fecha:** 27 de Octubre, 2025

## Análisis del Caso Organizacional y Propuesta

(Contexto de la Evaluación 1)

1. Nombre y Descripción: Everlast Chile, retail de equipamiento deportivo de combate (boxeo, MMA, fitness).
2. Problema/Desafío: Alta carga operativa en soporte al cliente debido a consultas técnicas repetitivas sobre productos, generando largos tiempos de respuesta, experiencia inconsistente y pérdida de ventas.
3. Objetivos: Desarrollar un "Asistente de Ventas Virtual" (chatbot IA) para:
   * Proporcionar recomendaciones personalizadas.
   * Responder preguntas frecuentes (productos, tallas, políticas) 24/7.
   * Reducir carga del equipo de soporte.
4. Datos Disponibles: Se simula acceso a información interna en formato .md: descripciones de productos, guías de tallas y políticas comerciales.
5. Restricciones: El chatbot debe responder únicamente con la información proporcionada, sin alucinaciones.
6. Motivación para RAG: Un LLM estándar desconoce los detalles específicos de Everlast Chile. La arquitectura RAG (Recuperación Aumentada por Generación) es esencial para conectar el LLM a una base de conocimiento controlada, garantizando respuestas precisas.

## (B) Diseño e Implementación del Agente (IL2.1)

Para cumplir con los objetivos, se implementó un agente funcional con una arquitectura basada en LangChain y Streamlit. Este agente integra herramientas de consulta y razonamiento para automatizar el flujo de atención al cliente.

El agente tiene acceso a dos herramientas principales, definidas en tools\_everlast.py:

1. Herramienta de Consulta (BusquedaDocumentosEverlast): Esta es la herramienta principal de RAG. Su función es buscar en la base de conocimiento interna de Everlast (productos, tallas, políticas). El pipeline RAG implementado sigue estos pasos:
   * Carga y División (Chunking): Los documentos .md se cargan y dividen en fragmentos (chunks).
   * Vectorización (Embeddings): Cada chunk se convierte en un vector numérico usando text-embedding-3-small.
   * Almacenamiento (Vector Store): Los vectores se almacenan en un índice FAISS para búsqueda rápida. Este índice se crea *una sola vez* ejecutando create\_vectorstore.py.
   * Recuperación: Cuando el agente usa esta herramienta, la consulta del usuario se vectoriza y FAISS devuelve los 3 chunks más relevantes.
2. Herramienta de Razonamiento (CalculadoraSimple): Esta herramienta otorga al agente la capacidad de realizar cálculos matemáticos básicos. Es esencial para tareas como:
   * Cálculo de descuentos (ej. 50000 \* 0.8).
   * Conversiones de unidades (ej. de libras a kg).
   * Cálculos de precios con IVA.

## Configuración de Memoria y Recuperación (IL2.2)

El agente gestiona la memoria de dos formas para asegurar la coherencia en flujos prolongados:

1. Memoria a Largo Plazo (Recuperación de Contexto Semántico): Esta memoria es el conocimiento persistente del agente. Está implementada a través del pipeline RAG y el índice FAISS.
   * Función: Almacena todo el conocimiento de productos, tallas y políticas de Everlast de forma vectorizada.
   * Recuperación (IL2.2): Proporciona la "recuperación de contexto semántico". Cuando el agente necesita saber sobre un producto, no usa su memoria de corto plazo, sino que "recupera" el contexto semántico relevante desde FAISS.
2. Memoria a Corto Plazo (Memoria de Contenido): Esta memoria gestiona el contexto de la conversación activa. Está implementada en la clase MemoriaSimple dentro de agente\_principal.py.
   * Función: Asegura que el agente pueda responder preguntas de seguimiento (ej. "y de qué colores tienes ese guante?").
   * Justificación (Buffer + Window): La clase MemoriaSimple actúa como una "Buffer Memory" (memoria de búfer), ya que almacena el historial completo de la sesión en self.historial.
   * Sin embargo, para optimizar el rendimiento y el costo de tokens, se aplica una lógica de "Window Memory" (memoria de ventana) al momento de consultar al LLM, enviando solo los últimos 10 mensajes (obtener\_historial(ultimos\_n=10)). Esta estrategia híbrida mantiene la fluidez de la conversación (cumpliendo IL2.2) de manera eficiente.

## (D) Planificación y Toma de Decisiones (IL2.3)

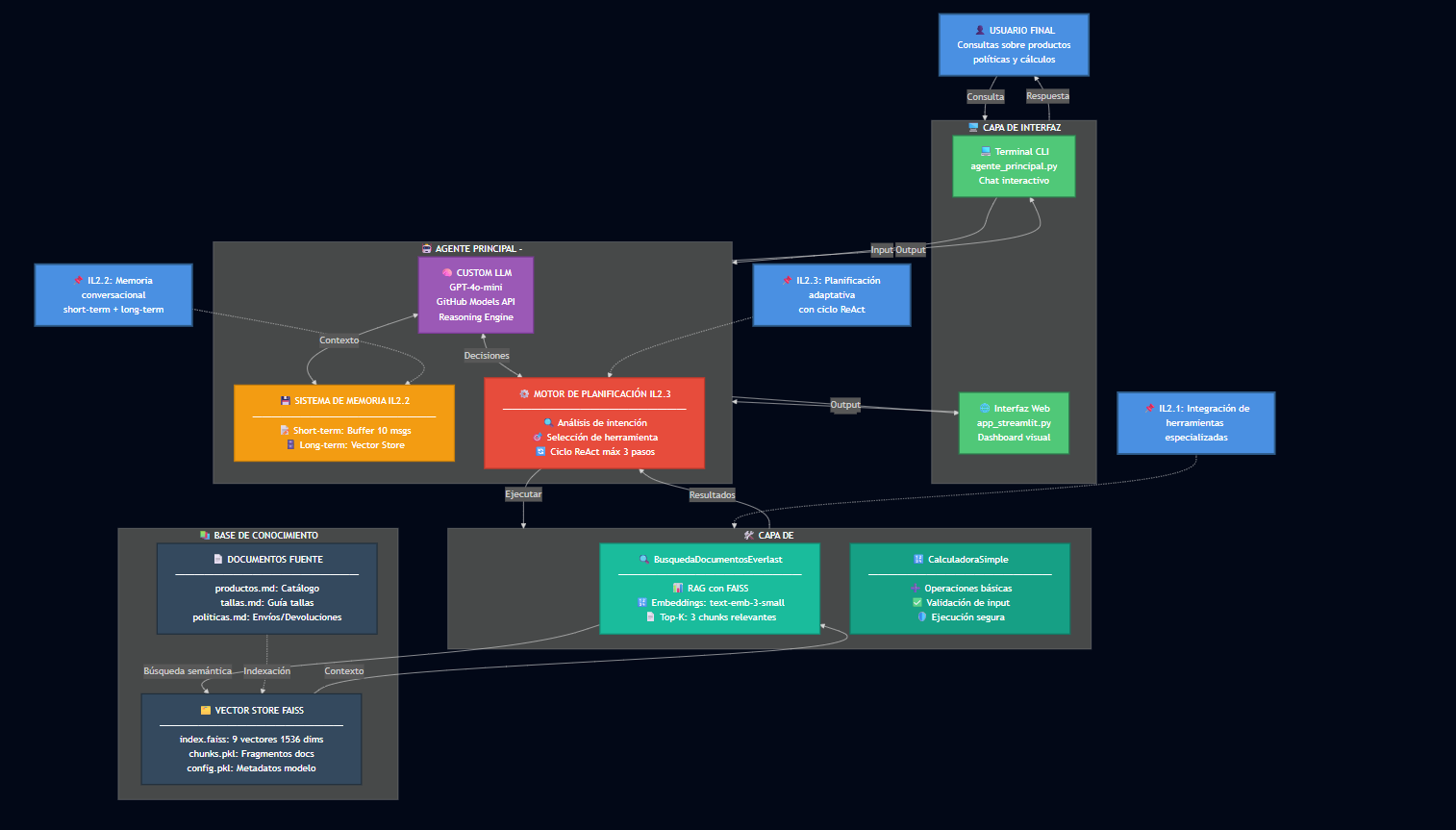
El agente implementa una estrategia de planificación ReAct (Reasoning + Acting) para gestionar tareas con múltiples etapas y condiciones cambiantes. Este ciclo se gestiona en la función procesar de agente\_principal.py.

El flujo de decisión es el siguiente:

1. Razonamiento (Reasoning): El agente recibe la consulta del usuario y el historial de corto plazo. Se le presenta un Prompt de Sistema (system\_prompt en agente\_principal.py) que actúa como su manual de instrucciones. Este prompt le indica qué herramientas tiene y *cuándo* usarlas (ej. "si la pregunta es sobre productos... USA buscar\_documentos").
2. Acción (Acting): Basado en su razonamiento, el agente decide si debe responder directamente (para un saludo) o si debe usar una herramienta. Si elige una herramienta, genera el INPUT específico para ella (ej. INPUT: política de devolución).
3. Observación: El agente ejecuta la herramienta (ej. \_ejecutar\_herramienta) y recibe una "Observación" (el resultado de la herramienta, como los chunks de texto o el resultado del cálculo).
4. Ciclo ReAct: El agente vuelve al Paso 1, pero esta vez con la "Observación" añadida al contexto. Razona nuevamente: "¿Tengo suficiente información para responder?"
   * Si la respuesta es sí, genera la RESPUESTA: final.
   * Si necesita más información (ej. calcular un descuento *después* de encontrar un precio), puede volver a ejecutar otra herramienta, demostrando planificación multi-etapa.

## (E) Arquitectura de la Solución y Documentación (IL2.4)

## Diagrama de Arquitectura



La arquitectura es un sistema RAG orquestado por LangChain y presentado a través de una interfaz web Streamlit. El usuario interactúa vía Streamlit. La consulta va a la cadena RetrievalQA (implementada como la herramienta BusquedaDocumentosEverlast), que usa el modelo de Embeddings para vectorizarla y buscar chunks similares en FAISS. Estos chunks (contexto) y la consulta original se envían al LLM (gpt-4o) para generar la respuesta, que vuelve a Streamlit.

Justificación de Elección de Componentes (IL2.4 / IE8)

* LangChain: Framework líder para RAG, simplifica la orquestación con componentes modulares.
* OpenAI/GitHub Models API (gpt-4o, text-embedding-3-small): Modelos potentes y accesibles vía credenciales del curso.
* FAISS: Base vectorial en memoria, eficiente y simple para prototipos. Ideal para no depender de servicios externos durante la evaluación.
* Streamlit: Permite crear UIs interactivas rápidamente, ideal para demostración sin ser experto web.
* Gestión de Credenciales: Se utiliza un archivo .env para almacenar claves API y URLs sensibles. Se configuró un .gitignore que instruye a Git para ignorar .env y venv/, previniendo exposición de credenciales en GitHub.

## (F)

## Referencias

* *FAISS*. (s.f.). Facebook AI Research. Recuperado de <https://faiss.ai/>
* *LangChain*. (s.f.). LangChain Inc. Recuperado de <https://www.langchain.com/>
* *Streamlit*. (s.f.). Streamlit Inc. Recuperado de <https://streamlit.io/>
* *OpenAI*. (s.f.). OpenAI, L.L.C. Recuperado de <https://openai.com/>
* *Python-dotenv*. (s.f.). Recuperado de <https://pypi.org/project/python-dotenv/>