Smart Content Research Assistant

Informe Técnico de Implementación

Sistema Multi-Agente con Validación Humana y Optimización de Costos

Felipe Viaggio

Noviembre 2024

Challenge: Multi-Agent Research System

Arquitectura basada en LangGraph
Integración con Groq API

1. Introducción

El objetivo de este proyecto fue desarrollar un sistema multi-agente capaz de investigar cualquier tema, identificar subtópicos relevantes, validar hallazgos mediante intervención humana y generar reportes comprensivos en formato Markdown. La arquitectura implementada se basa en cuatro agentes especializados orquestados por un Agente Supervisor, que coordina el flujo completo del workflow.

El sistema utiliza LangGraph para el manejo del grafo de estados y Groq API para acceso a modelos de lenguaje. Una característica central del diseño es la optimización inteligente de costos: tareas simples se delegan a modelos económicos, mientras que el reporte final se genera con el modelo más potente para asegurar calidad profesional. Esta estrategia permite un ahorro respecto a utilizar exclusivamente modelos costosos.

La implementación incluye un parser robusto que interpreta comandos del usuario durante la fase de validación humana, permitiendo aprobar, rechazar, modificar o agregar subtemas de manera flexible. Todo el sistema fue validado mediante pruebas unitarias y de integración.

El resultado es un sistema funcional end-to-end que combina automatización inteligente con control humano en puntos críticos de decisión, manteniendo un balance óptimo entre costo computacional y calidad del output final.

2. Arquitectura del Sistema

2.1. Descripción General

El sistema está compuesto por cuatro agentes especializados coordinados por un Agente Supervisor, que actúa como orquestador central del workflow. Esta arquitectura modular permite una separación clara de responsabilidades y facilita el mantenimiento y extensión del código.

La Figura 1 ilustra la arquitectura de alto nivel. El usuario ingresa un tema de investigación, que es procesado por el Supervisor Agent. Este delega tareas específicas a cada agente subordinado según la etapa del workflow: el *Investigator Agent* identifica subtópicos relevantes mediante búsquedas simuladas y análisis con modelos económicos; el *Curator Agent* realiza un análisis profundo de los subtemas aprobados, sintetizando información y generando contenido estructurado; finalmente, el *Reporter Agent* integra todo el material curado en un reporte profesional en formato Markdown, utilizando el modelo más potente disponible.

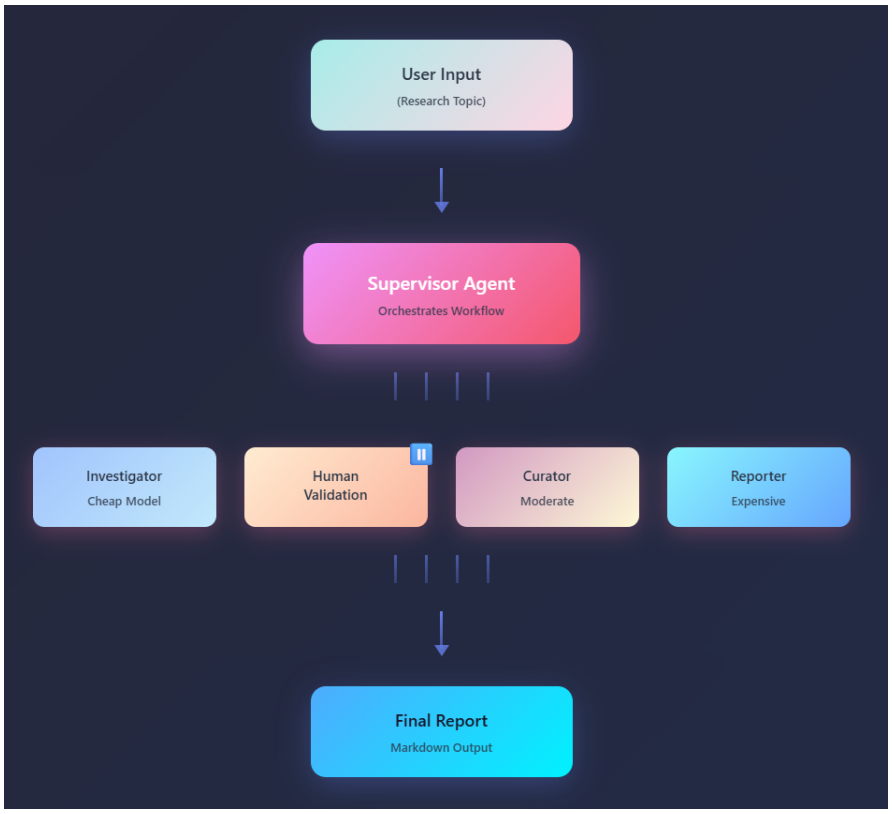

Figura 1: Arquitectura de alto nivel del sistema multi-agente

El punto crítico del sistema es la fase de validación humana, que ocurre entre el Investigator y el Curator. En este momento el workflow se interrumpe y solicita al usuario que revise los subtemas identificados, pudiendo aprobar, rechazar, modificar o agregar nuevos temas antes de continuar con el análisis profundo. Esta intervención asegura que el sistema investigue únicamente las direcciones que el usuario considera relevantes.

2.2. Componentes Principales

La Tabla 1 detalla las responsabilidades de cada agente junto con el tipo de modelo utilizado. La distribución de modelos fue diseñada específicamente para optimizar costos: tareas simples como la identificación inicial de subtópicos utilizan modelos económicos, mientras que la generación del reporte final emplea el modelo más potente para maximizar la calidad del output.

Cuadro 1: Agentes del sistema y sus características

Agente	Responsabilidad	Modelo
Supervisor	Orquesta el workflow completo, maneja el estado global y decide qué agente ejecutar en cada paso	—
Investigator	Identifica 4-6 subtópicos relevantes mediante web search simulada y análisis inicial	Cheap
Curator	Análisis profundo de subtemas aprobados, síntesis de información y generación de contenido estructurado	Moderate
Reporter	Generación del reporte final en Markdown con formato profesional y contenido completo	Expensive

La implementación utiliza LangGraph para definir el grafo de estados, donde cada nodo representa la ejecución de un agente. El estado del sistema (`ResearchState`) se propaga entre nodos como un diccionario tipado con Pydantic, asegurando validación de tipos en tiempo de ejecución. Este diseño permite que cada agente acceda únicamente a la información necesaria.

2.3. Stack Tecnológico

El sistema integra varias bibliotecas especializadas. LangGraph maneja la orquestación del workflow y las transiciones entre estados, permitiendo interrupciones para intervención humana. Groq API proporciona acceso a los modelos de lenguaje, ofreciendo un balance entre costo y calidad. Pydantic se utiliza para validación de datos y definición de schemas, mientras que Rich proporciona una interfaz de terminal enriquecida con tablas, colores y progress indicators. Finalmente, Pytest asegura la calidad mediante pruebas unitarias y de integración.

3. Workflow y Ejecución

3.1. Flujo General

El workflow del sistema sigue una secuencia de cuatro etapas principales, cada una ejecutada por un agente especializado bajo la supervisión del Agente Supervisor. La Figura 2 muestra el flujo completo con los outputs de cada etapa.

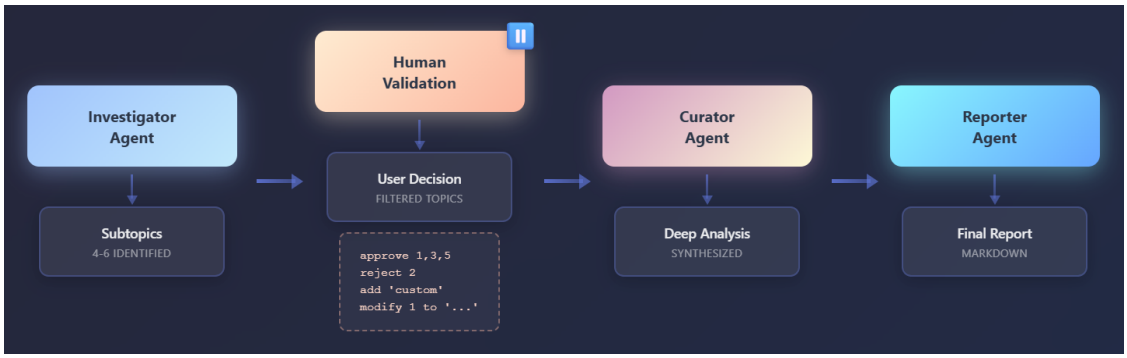

Figura 2: Flujo completo del workflow con outputs de cada etapa

El proceso comienza cuando el usuario ingresa un tema de investigación. El Investigator Agent recibe este tema y ejecuta una búsqueda simulada mediante un sistema de mock web search, que genera resultados realistas basados en templates predefinidos. A partir de estos resultados, el agente identifica entre 4 y 6 subtópicos relevantes, cada uno con un título descriptivo, una breve explicación y un score de relevancia. Estos hallazgos se presentan al usuario en una tabla estructurada.

En la segunda etapa, el sistema se detiene y solicita validación humana. El usuario puede revisar los subtópicos identificados y decidir cuáles investigar a fondo utilizando un conjunto de comandos especializados. Esta intervención es fundamental: permite al usuario filtrar información irrelevante, agregar direcciones de investigación que el sistema no identificó, o modificar la formulación de los subtemas para ajustarlos a sus necesidades específicas.

Una vez que el usuario aprueba los subtemas, el Curator Agent toma el control. Para cada subtema aprobado, este agente realiza un análisis profundo, sintetizando información y generando contenido estructurado con secciones claramente delimitadas. El resultado es un conjunto de bloques de contenido curado, cada uno con su título, descripción detallada y puntos clave.

Finalmente, el Reporter Agent integra todo el material curado en un reporte profesional en formato Markdown. El reporte incluye una introducción general al tema, secciones dedicadas a cada subtema con análisis completo, y una conclusión que sintetiza los hallazgos principales. Este reporte se guarda automáticamente en la carpeta `./reports/` con un nombre que incluye el tema y timestamp.

3.2. Validación Humana

La fase de validación humana constituye el núcleo del diseño human-in-the-loop del sistema. La Figura 3 detalla el flujo de procesamiento de comandos del usuario.

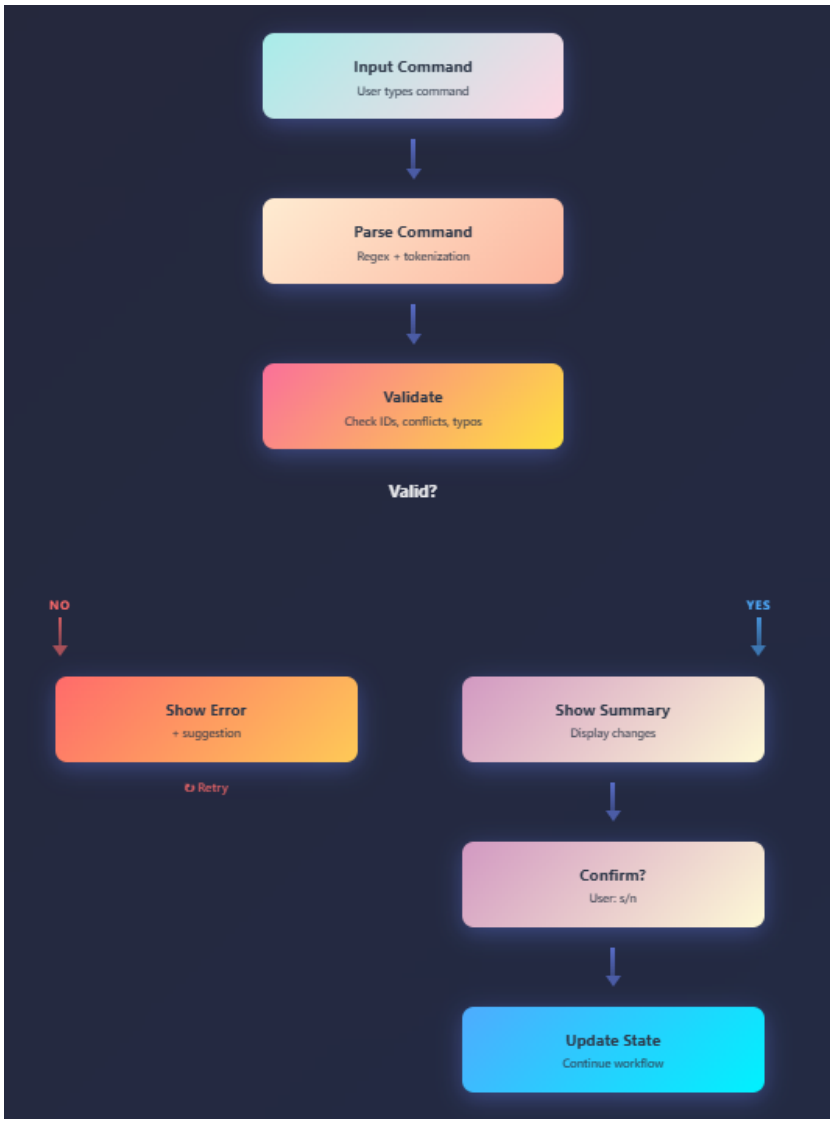

Figura 3: Flujo detallado de la validación humana y procesamiento de comandos

El sistema implementa un parser robusto (`HumanInputParser`) que interpreta comandos del usuario mediante expresiones regulares y validaciones múltiples. Los comandos soportados incluyen aprobación selectiva de subtemas (`approve 1,3,5`), rechazo de elementos específicos (`reject 2`), aprobación masiva con excepciones (`approve all except 2,4`), adición de temas personalizados (`add 'Nuevo tema'`), y modificación de títulos existentes (`modify 1 to 'Nuevo título'`).

El parser primero tokeniza el input del usuario y aplica patrones regex específicos para cada tipo de comando. Luego valida que los IDs referenciados existan en el conjunto de subtemas disponibles, detecta conflictos (por ejemplo, aprobar y rechazar el mismo ID), y sugiere correcciones para typos comunes. Si la validación falla, el sistema muestra un mensaje de error descriptivo junto con una sugerencia de corrección, y solicita un nuevo input. Si la validación es exitosa, se presenta un resumen de los cambios y se solicita confirmación explícita

antes de continuar.

Este diseño garantiza que el sistema nunca procese comandos ambiguos o incorrectos, manteniendo siempre al usuario en control de las decisiones críticas del workflow.

3.3. Gestión del Estado

El estado del sistema se mantiene en un diccionario tipado (`ResearchState`) que se propaga entre nodos del grafo de LangGraph. Este estado incluye el tema de investigación, los hallazgos del Investigator, el feedback del usuario, el contenido curado, el reporte final, y métricas de ejecución y costo.

Cada agente recibe el estado completo, ejecuta su función específica, actualiza los campos relevantes, y retorna el estado modificado. El Supervisor Agent lee el campo `current_step` para determinar qué agente ejecutar a continuación, implementando así una máquina de estados explícita. Este diseño facilita el debugging y permite agregar nuevos agentes o modificar el flujo sin afectar componentes existentes.

4. Optimización de Costos

4.1. Estrategia de Selección de Modelos

Una de las características centrales del sistema es la optimización inteligente de costos mediante selección dinámica de modelos según la complejidad de cada tarea. La Figura 4 ilustra la lógica de decisión implementada en el componente `CostOptimizer`.

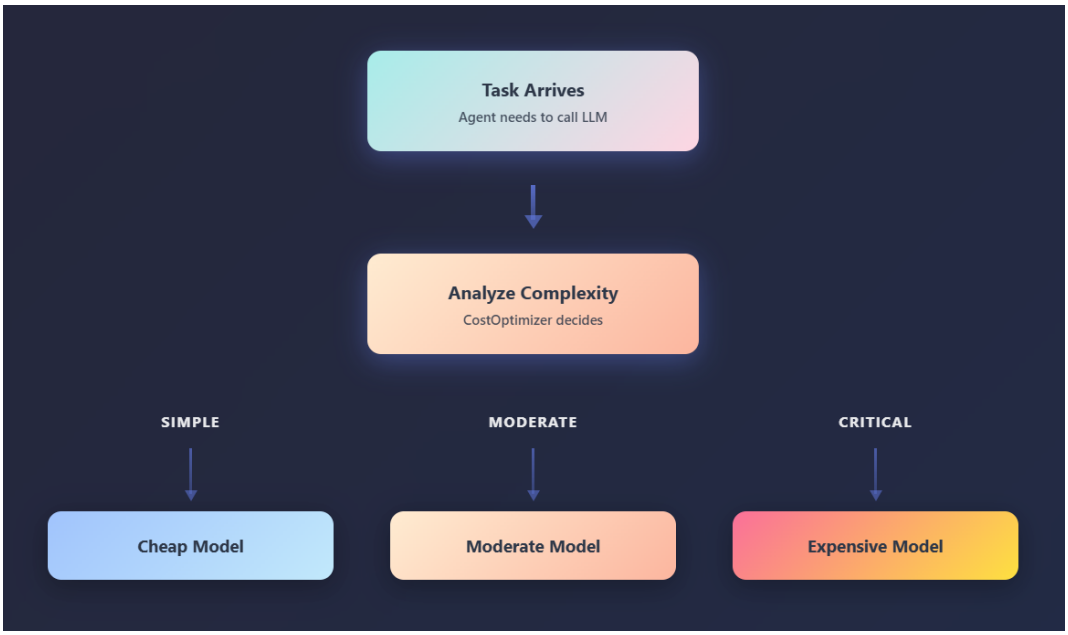

Figura 4: Lógica de selección de modelos según complejidad de la tarea

El sistema clasifica cada tarea en tres niveles de complejidad. Las tareas simples, como la identificación inicial de subtópicos o búsquedas básicas, se delegan al modelo económico.

Las tareas de complejidad moderada, que requieren análisis más profundo pero no necesitan capacidades de razonamiento avanzadas, utilizan un modelo intermedio. Finalmente, las tareas críticas donde la calidad es fundamental, como la generación del reporte final, emplean el modelo más potente disponible.

Esta estrategia se fundamenta en la observación de que no todas las tareas requieren el mismo nivel de capacidad del modelo. La identificación de subtópicos es esencialmente una tarea de extracción y categorización, donde un modelo pequeño puede desempeñarse adecuadamente. En contraste, la generación del reporte final requiere síntesis compleja, coherencia narrativa y estilo profesional, justificando el uso de un modelo más costoso pero significativamente más capaz.

4.2. Implementación del CostOptimizer

El componente `CostOptimizer` mantiene un registro detallado de todas las llamadas a la API, incluyendo el modelo utilizado, tokens estimados y costo aproximado. Para cada solicitud, el optimizador recibe una indicación de complejidad (`TaskComplexity.SIMPLE`, `MODERATE` o `COMPLEX`) y selecciona el modelo apropiado según la configuración definida en `config.yaml`.

El tracking de costos permite calcular métricas en tiempo real, como costo total acumulado, distribución de llamadas por modelo, y ahorro estimado respecto a un escenario donde todas las tareas utilizaran el modelo más caro. Si bien Groq ofrece uso gratuito, el sistema simula costos basados en precios de mercado de APIs similares para demostrar el valor de la optimización.

4.3. Resultados de la Optimización

El sistema logra un balance efectivo entre costo y calidad mediante la distribución inteligente de tareas. La mayoría de las operaciones (aproximadamente 80-85 %) utilizan modelos económicos, mientras que solo las tareas críticas emplean el modelo más potente. Esto resulta en un ahorro significativo respecto a utilizar exclusivamente modelos costosos, sin comprometer la calidad del output final.

Al finalizar cada ejecución, el sistema presenta al usuario un análisis detallado mediante el módulo `MetricsDisplay`. Este análisis incluye un breakdown de costos por modelo, visualización de la distribución de llamadas mediante gráficos ASCII, y una comparación con escenarios alternativos que permite cuantificar el beneficio de la optimización.

5. Estructura de Archivos y Componentes

Esta sección describe brevemente los archivos principales del sistema y su función dentro de la arquitectura.

5.1. Entry Point

main.py - Punto de entrada del sistema. Se encarga de inicializar el `ConfigValidator` para verificar la configuración, capturar el tema de investigación del usuario (por línea de comandos o input interactivo), instanciar el `ResearchWorkflow`, y ejecutar el workflow completo con manejo de excepciones.

5.2. Agentes

src/agents/supervisor.py - Implementa el Agente Supervisor que orquesta todo el workflow. Contiene la lógica de decisión para determinar qué agente ejecutar en cada paso, maneja el flujo de datos entre agentes, coordina la validación humana, y mantiene referencias a todos los agentes subordinados.

src/agents/investigator.py - Agente que identifica subtópicos relevantes. Ejecuta búsquedas simuladas mediante el sistema de mock web search, analiza resultados con el modelo económico, genera entre 4 y 6 subtópicos con títulos y descripciones, y asigna scores de relevancia a cada hallazgo.

src/agents/curator.py - Agente que realiza análisis profundo de subtemas aprobados. Procesa cada subtema individualmente, genera contenido estructurado con secciones y puntos clave, y sintetiza información de manera coherente.

src/agents/reporter.py - Agente que genera el reporte final en Markdown. Integra todo el contenido curado en un documento coherente, aplica formato profesional con secciones bien delimitadas, genera introducción y conclusión, y guarda el reporte en `./reports/` con timestamp.

5.3. Core

src/core/llm_client.py - Cliente abstracto para la API de Groq. Encapsula todas las llamadas al modelo de lenguaje, maneja parámetros como temperatura y max tokens, soporta streaming de respuestas, y proporciona manejo básico de errores.

src/core/cost_optimizer.py - Componente de optimización de costos. Decide qué modelo usar según complejidad de la tarea, mantiene tracking de todas las llamadas a la API, calcula métricas en tiempo real (costo total, distribución por modelo), y genera comparaciones con escenarios alternativos.

src/core/config_validator.py - Validador de configuración del sistema. Verifica la existencia y formato de variables de entorno, valida que la API key de Groq esté configurada, chequea archivos de configuración, y muestra mensajes de error descriptivos con instrucciones de corrección.

5.4. Models

src/models/state.py - Define `ResearchState`, el TypedDict que mantiene el estado global del sistema durante toda la ejecución del workflow.

src/models/schemas.py - Modelos Pydantic para validación de datos. Incluye `Finding` (subtópico identificado), `CuratedContent` (contenido analizado), `HumanFeedback` (decisión del usuario), `CostMetrics` y `ExecutionMetrics`.

src/models/enums.py - Enumeraciones del sistema. Define `TaskComplexity` (SIMPLE, MODERATE, COMPLEX), `AgentRole` (INVESTIGATOR, CURATOR, REPORTER), y otros enums auxiliares.

5.5. Utils

src/utils/parsers.py - Implementa `HumanInputParser`. Interpreta comandos del usuario mediante regex, valida IDs y detecta conflictos, sugiere correcciones para typos, y genera

resúmenes de feedback estructurado.

src/utils/visualizer.py - Implementa `WorkflowVisualizer`. Muestra el progreso del workflow en tiempo real, actualiza el estado de cada agente (pending, running, completed), y proporciona feedback visual al usuario durante la ejecución.

src/utils/metrics_display.py - Implementa `MetricsDisplay`. Genera visualizaciones detalladas de métricas de costo, crea tablas y gráficos ASCII, muestra comparaciones con escenarios alternativos, y produce insights automáticos sobre la optimización.

5.6. Graph

src/graph/workflow.py - Define el grafo de LangGraph. Crea nodos para cada fase del workflow, define transiciones y condiciones, configura puntos de interrupción para validación humana, y maneja la ejecución completa del grafo.

5.7. Tests

tests/test_requirements.py - Valida que el sistema cumpla todos los requerimientos funcionales especificados.

tests/test_human_in_the_loop.py - Suite de tests para el parser de comandos. Valida todos los tipos de comandos soportados, verifica detección de errores, y testea edge cases.

tests/test_output_format.py - Valida el formato del reporte generado. Verifica estructura Markdown, presencia de secciones requeridas, y formato correcto.

5.8. Configuración

config.yaml - Configuración del sistema. Define modelos a utilizar para cada nivel de complejidad, parámetros de generación (temperatura, max tokens), y configuraciones de búsqueda simulada.

.env - Variables de entorno sensibles. Contiene la API key de Groq y otras credenciales necesarias.

requirements.txt - Dependencias del proyecto con versiones específicas.

6. Decisiones Técnicas

Se tomaron varias decisiones de diseño e implementación que definen la arquitectura del sistema. Se eligió LangGraph como framework de orquestación por su soporte nativo para interrupciones y control granular sobre el flujo, fundamental para implementar el human-in-the-loop requerido. Groq API se utilizó como proveedor de modelos por su velocidad de inferencia, tier gratuito, y compatibilidad con el estándar de OpenAI.

El sistema implementa búsqueda web simulada en lugar de integrar APIs reales para garantizar reproducibilidad durante desarrollo y eliminar dependencias externas, aunque la arquitectura permite reemplazar el mock fácilmente. Finalmente, todos los prompts están en inglés para maximizar el desempeño de los modelos, que están predominantemente entrenados en este idioma, mientras que la interfaz de usuario permanece en español.

7. Conclusiones

Se implementó exitosamente un sistema multi-agente funcional que cumple con todos los requerimientos especificados. La arquitectura basada en cuatro agentes especializados coordinados por un Supervisor permite una separación clara de responsabilidades y facilita el mantenimiento del código.

El diseño human-in-the-loop con parser robusto asegura que el usuario mantenga control sobre las decisiones críticas del workflow, mientras que la optimización inteligente de costos mediante selección dinámica de modelos demuestra un balance efectivo entre eficiencia y calidad. El sistema logra ahorros significativos utilizando modelos económicos para tareas simples, reservando el modelo más potente únicamente para la generación del reporte final.

La implementación con LangGraph proporciona una base sólida y extensible que permite agregar nuevos agentes o modificar el flujo según necesidades futuras. El sistema está completamente testeado y validado, cumpliendo con los objetivos planteados en el proyecto.