Informe Técnico: Arquitectura de Agente de Generación de Noticias

Fecha: 14 de diciembre de 2025 **Autor:** Iñaki Cejudo **Tecnología:** LangGraph, LangChain, OpenAI (GPT-4o), Pydantic v2

1. Resumen Ejecutivo

Este documento detalla la arquitectura técnica de un Agente de IA diseñado para la automatización de publicaciones en redes sociales. El sistema ingiere URLs de noticias, procesa el contenido mediante LLMs y genera salidas estructuradas (JSON) listas para consumo vía API. La arquitectura implementa un flujo *Human-in-the-loop* (Humano en el bucle), permitiendo ciclos de revisión y edición antes de la publicación final.

2. Arquitectura del Sistema

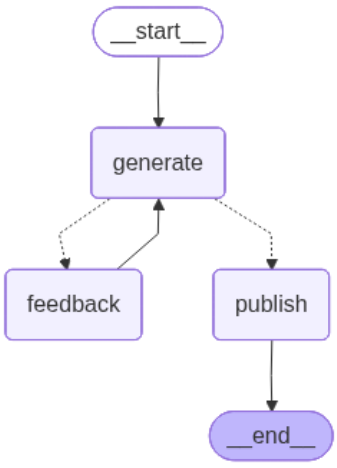
El núcleo del sistema se basa en **LangGraph**, utilizando un **Grafo de Estado (StateGraph)** cíclico en lugar de una cadena lineal (DAG). Esto permite que el agente tenga "memoria" y pueda retroceder a estados anteriores para corregir errores o refinar contenido.

2.1 Componentes Principales

1. **Estado Global (PostState):** Un TypedDict que actúa como la memoria a corto plazo del agente, persistiendo entre nodos. Almacena:
    - messages: Historial de conversación.
    - web_content: Caché del contenido crudo de la noticia (evita múltiples llamadas HTTP).
    - last_post: Objeto estructurado (SocialPost) con la última versión generada.
2. **Capa de Inferencia Estructurada:** Uso de gpt-4o con with_structured_output para garantizar que la salida cumpla estrictamente con el esquema Pydantic.
3. **Router de Intención (Pre-Grafo):** Funciones deterministas y ligeras que clasifican la intención del usuario antes de invocar el grafo costoso.

2.2 Diagrama del Grafo (Esquema Monolítico)

A diferencia de un sistema multi-agente distribuido, aquí centralizamos la lógica cognitiva en un nodo robusto, simplificando la orquestación.



Fragmento de código

```
graph TD
    UserInput([Entrada Usuario]) --> Router{Clasificación}
    Router -- CHAT --> DirectResp[Respuesta Directa]
    Router -- NEW_POST --> Init[Inicializar Estado]

    subgraph "LangGraph (Agente)"
        Init --> Generate[Nodo: Generate (Lógica Central)]
        Generate --> Validation{Validación Humana}

        Validation -- Rechazado --> Feedback[Nodo: Feedback]
        Feedback -- Instrucciones de cambio --> Generate

        Validation -- Aprobado --> Publish[Nodo: Publish]
    end

    Publish -- POST JSON --> API[API Externa / DB]
    API --> Fin([Fin del Flujo])
```

3. Decisiones de Diseño Clave

3.1 Arquitectura Monolítica vs. Sistema Multi-Agente

Una de las decisiones arquitectónicas más críticas fue determinar cómo distribuir las responsabilidades.

- **Opción Descartada (Multi-Agente):** Se evaluó crear un enjambre de agentes especializados (un agente "Investigador" que usa Jina, un agente "Redactor" y un agente "Editor JSON").
    - *Problema:* Esto introduce latencia excesiva en la comunicación entre agentes y complejidad en la gestión del estado (pasar el contexto completo de un agente a otro).
- **Decisión Tomada (Monolítico con Herramientas):** Se optó por un **Agente Único con un Nodo Polimórfico (generate)**.
    - *Justificación:* Dado que el modelo GPT-4o es lo suficientemente potente para entender contexto, redacción y formato simultáneamente, centralizar la lógica en un solo nodo reduce la complejidad del grafo. El nodo generate actúa de forma diferente según el contexto (si es la primera vez, si es una edición o si es un cambio de fuente), pero mantiene acceso compartido a la memoria (web_content).

3.2 Selección del Framework (LangGraph vs. OpenAI Agents SDK)

Se priorizó el uso de **LangGraph** frente al enfoque gestionado del **OpenAI Agents SDK (Assistants API)** debido a la necesidad de un control determinista sobre el flujo de ejecución. Mientras que el SDK de OpenAI actúa como una "caja negra" donde la gestión del estado y la secuencia de pasos son opacas, LangGraph permite definir una arquitectura de grafos explícita. Esto resultó crucial para implementar un sistema **Human-in-the-loop** efectivo, permitiendo

detener la ejecución programáticamente para la validación del JSON y manipular directamente el estado (`PostState`) entre nodos, evitando además el *vendor lock-in* exclusivo con los modelos de OpenAI.

4. Principales Desafíos y Soluciones

4.1 Desafío: Alucinaciones en el Formato de Salida

Problema: En versiones iniciales, el LLM generaba texto libre. Al intentar conectar esto con una API, los parsers fallaban si el LLM añadía introducción ("Aquí tienes el JSON...") o cometía errores de sintaxis. **Solución: Structured Output con Pydantic.** Implementamos la clase SocialPost(BaseModel).

- Se fuerza al LLM a "rellenar un formulario" en lugar de escribir texto.
- Esto garantiza tipado estricto: title siempre es string, image_url es opcional pero validado.
- Permite usar post.model_dump() para enviar un payload limpio a la API sin necesidad de expresiones regulares (Regex).

4.2 Desafío: "Ruido" en el Contenido Web

Problema: Scrapear una web tradicional trae HTML sucio, menús de navegación, publicidad y scripts, lo que confunde al LLM y desperdicia tokens. **Solución: Integración con Jina AI.** Se externalizó la limpieza del contenido mediante la herramienta fetch_web que invoca a r.jina.ai.

- Esto convierte cualquier URL en un Markdown limpio y legible para el LLM.
- Mejora la calidad del resumen y reduce el coste de tokens de entrada significativamente.

4.3 Desafío: Ambigüedad en las Instrucciones de Edición

Problema: Cuando el usuario decía "cámbialo", el agente no sabía si debía reescribir el estilo o buscar otra noticia en el mismo texto. **Solución: Ingeniería de Prompts Dinámica + Funciones Helper.** Se crearon clasificadores intermedios (classify_feedback) que inyectan *prompts* específicos en el nodo generate:

- Si es **EDIT**: Se inyecta el last_post y se pide refactorización.
- Si es **RESELECT**: Se bloquea el uso de la noticia anterior y se fuerza a buscar en web_content.
- Si es **RESCRAPE**: Se activa la herramienta de búsqueda de URL.

5. Conclusión

La arquitectura actual ha evolucionado de un simple chatbot a un sistema de integración robusto. La combinación de **LangGraph** para el control de flujo y **Pydantic** para la validación de datos ha resuelto los problemas de fiabilidad, permitiendo que el agente funcione como un backend autónomo capaz de alimentar sistemas externos mediante API.