Tus agentes funcionan mejor en equipo.

Y **LangGraph** nos lo pone fácil

CONTEXTO

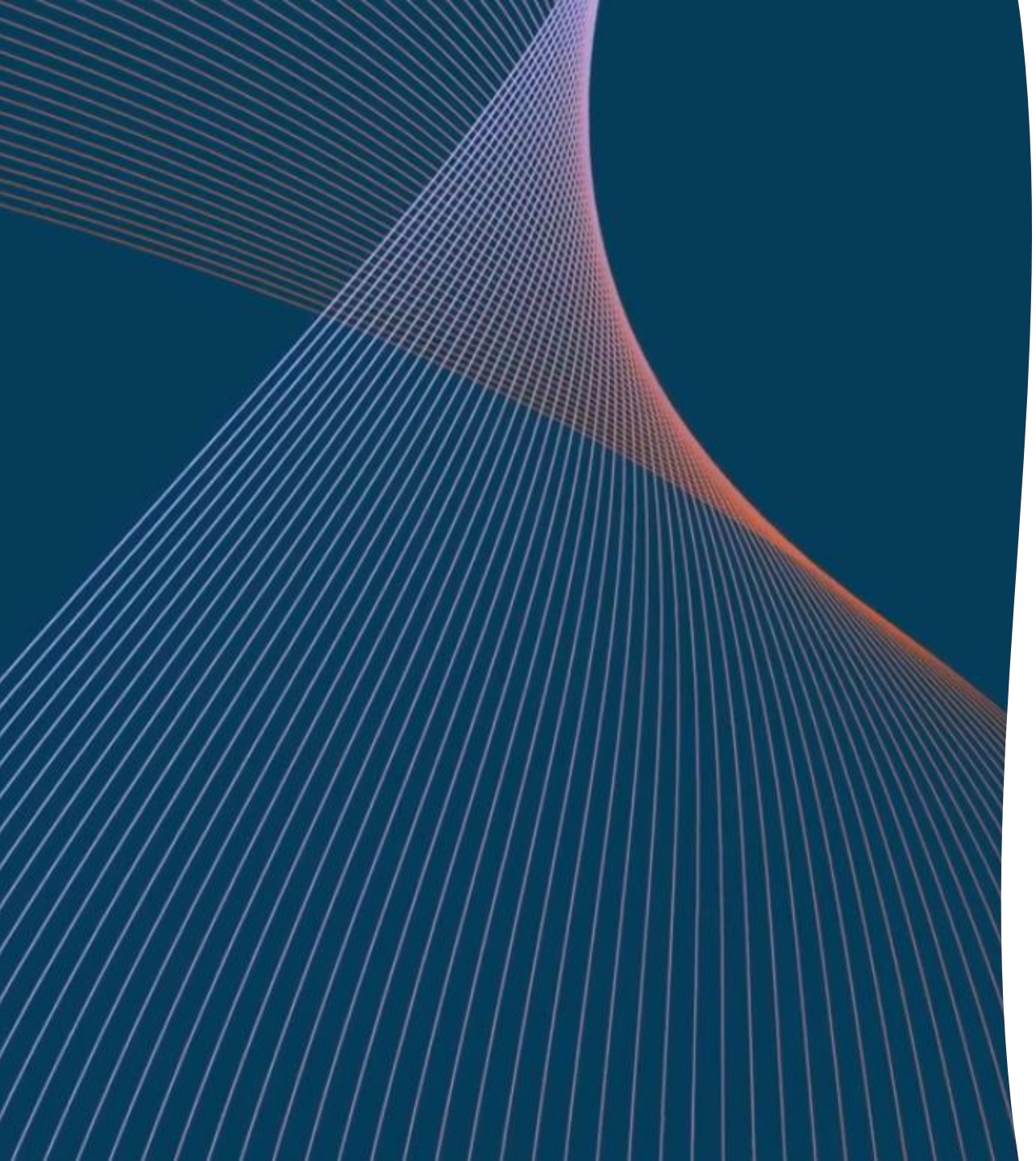
- Qué es un LLM?
- Qué es "prompting"?
- Qué es un agente?

¿Qué es un **LLM**?

- Large Language Model (Modelo de Lenguaje Grande)
- Es un tipo de inteligencia artificial entrenada con enormes cantidades de texto para entender y generar lenguaje humano.
- Ejemplos famosos: GPT, Claude, LLaMA, DeepSeek, Grok...

Limitaciones

- Carece de información en tiempo real
  - Se soluciona con el uso de RAGs: se trata de enriquecer el texto del prompt con información actualizada para ser usada como contexto.
- Puede alucinar
  - Cuanto más específico, mejor se comportará el modelo.
- Ineficiente para tareas que requieren varios pasos.
  - No se puede ser específico para varios casos en el mismo prompt.



¿Qué es 'prompting'?

- Es el **arte** de comunicarse e interactuar con el LLM.
- Se trata de explicar, "con pocas palabras", las instrucciones que debe seguir un LLM para llevar a cabo una tarea.
- Consejos para un buen prompt:
  - Establece el rol del modelo: Eres un experto en X y ayudas al usuario…
  - Sé claro y específico: evita ambigüedades.
  - Define el formato que quieres de salida: párrafos, lista, tabla, JSON…
  - Usa ejemplos (few-shot): 1-3 ejemplos de pregunta-respuesta.

Actúa como [rol] y realiza [tarea] en formato [tipo de salida], usando [estilo/tipo de lenguaje]. Limítate a [condiciones]. Aquí tienes el contexto: [contenido].

Estructura de un prompt
para Claude

1. Contexto de la tarea
2. Contexto del tono
3. Datos de contexto, documentos e imágenes
4. Descripción detallada de la tarea y reglas
5. Ejemplos
6. Historial de la conversación
7. Descripción inmediata de la tarea o solicitud
8. Pensar paso a paso / respira hondo
9. Formato de salida
10. Respuesta pre-rellenada (si existe)

Usuario

Actuarás como un coach de carrera con IA llamado Joe, creado por la empresa AdAstra Careers. Tu objetivo es dar consejos de carrera a los usuarios. Responderás a usuarios que estén en el sitio de AdAstra y que podrían confundirse si no respondes en el personaje de Joe.

Debes mantener un tono amigable de servicio al cliente.

Aquí está el documento de orientación de carrera que debes consultar al responder al usuario: <guide> {{DOCUMENT}} </guide>

Aquí hay algunas reglas importantes para la interacción:
- Mantente siempre en el personaje, como Joe, una IA de AdAstra Careers
- Si no estás seguro de cómo responder, di: "Lo siento, no entendí eso. ¿Podrías repetir la pregunta?"
- Si alguien pregunta algo irrelevante, di: "Lo siento, soy Joe y doy consejos de carrera. ¿Tienes una pregunta de carrera en la que pueda ayudarte hoy?"

Aquí tienes un ejemplo de cómo responder en una interacción estándar:
<example>
Usuario: Hola, ¿cómo fuiste creado y qué haces?
Joe: ¡Hola! Mi nombre es Joe, y fui creado por AdAstra Careers para dar consejos de carrera. ¿En qué puedo ayudarte hoy?
</example>

Aquí está el historial de la conversación (entre el usuario y tú) antes de la pregunta. Podría estar vacío si no hay historial: <history> {{HISTORY}} </history>
Aquí está la pregunta del usuario: <question> {{QUESTION}} </question>

¿Cómo respondes a la pregunta del usuario?

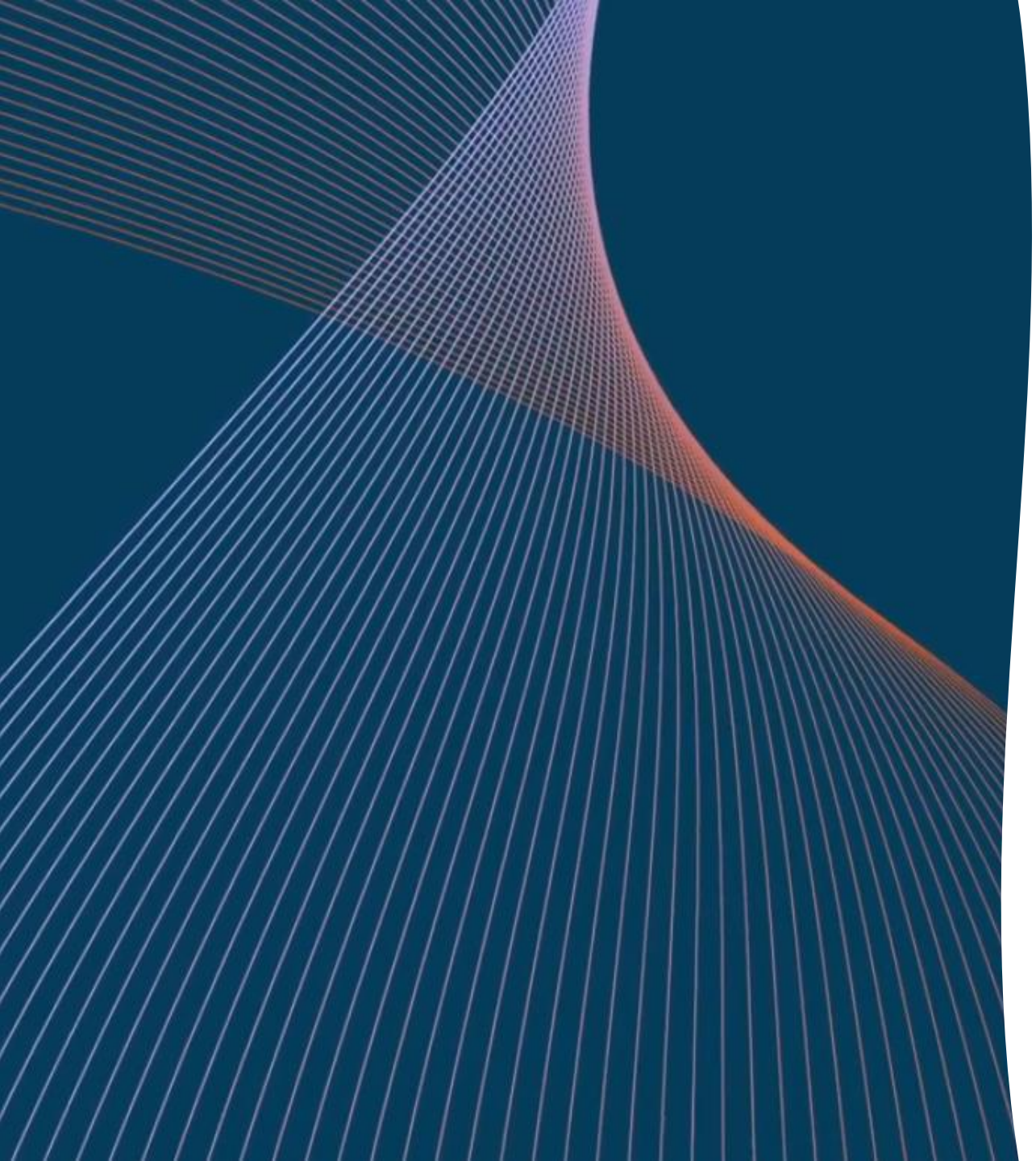
Piensa en tu respuesta antes de responder.

Pon tu respuesta en etiquetas <response></response>.

Asistente (prefill)

<response>

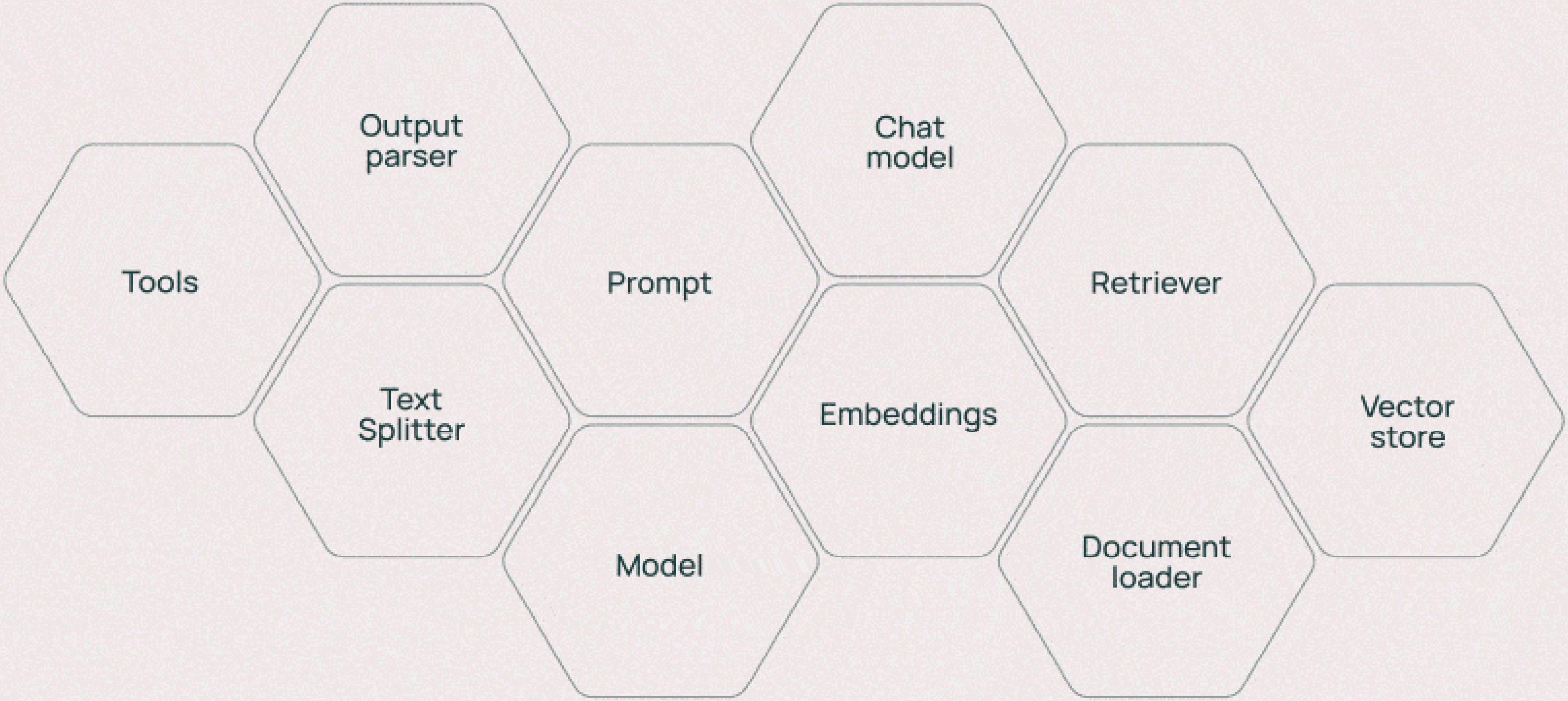
Traducido por @mouredev a partir del vídeo "Prompting 101 | Code w/ Claude" de Anthropic: https://youtu.be/ysPbXH0LpIE

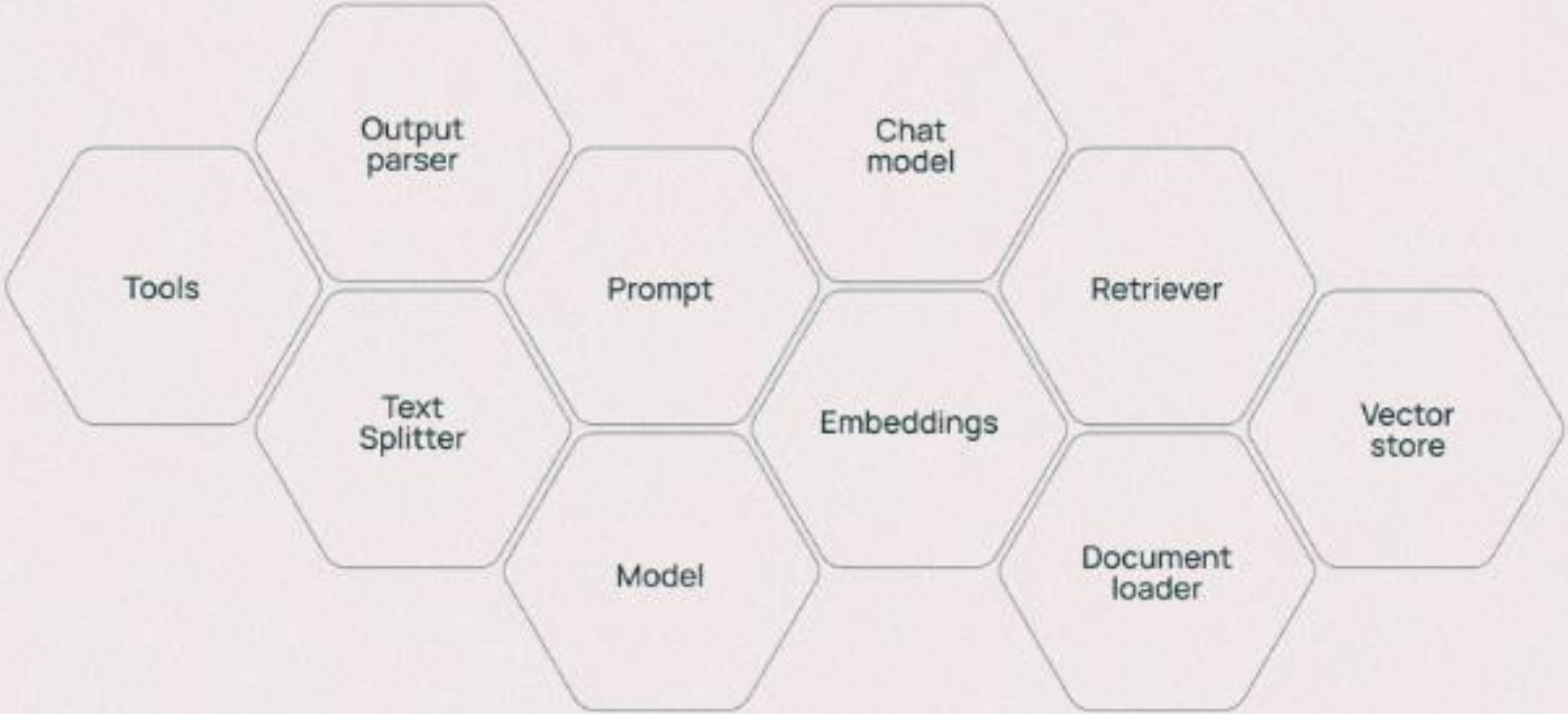


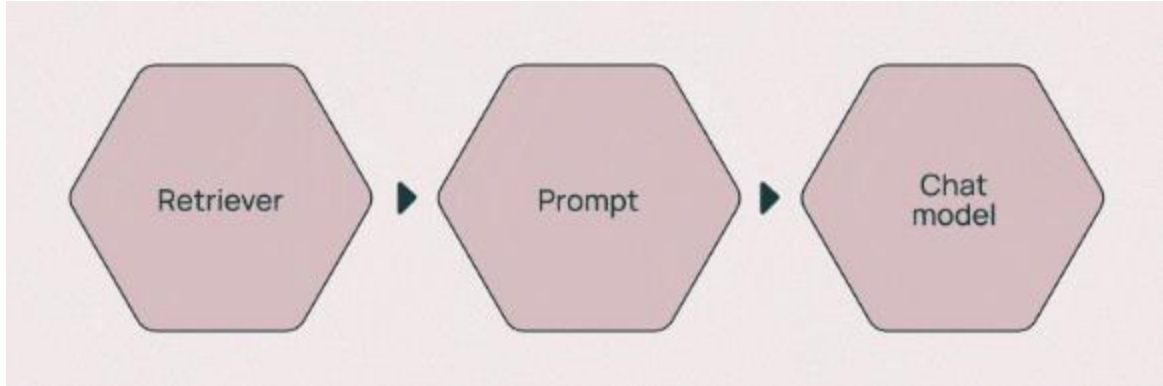
¿Qué es un agente?

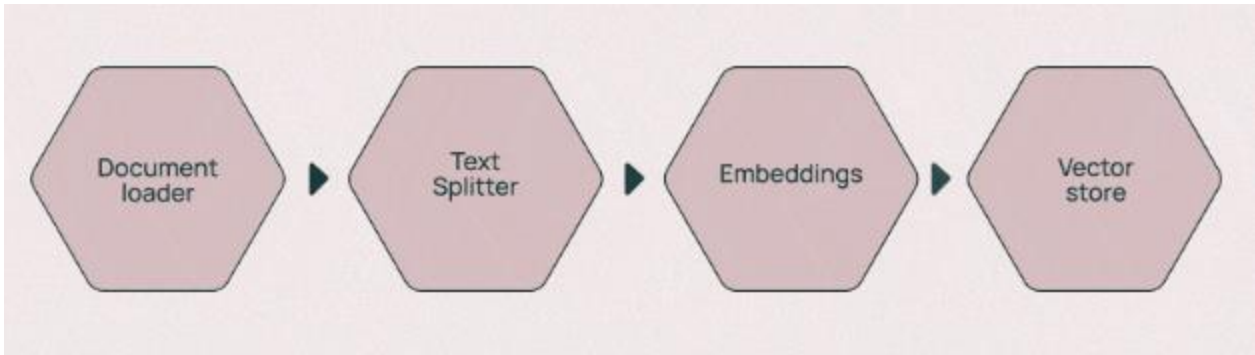
- Es un asistente autónomo que razona, toma decisiones y/o actúa para cumplir una tarea, usando el LLM como su "cerebro".
- Un agente es una entidad que:
  - Recibe una instrucción o una meta.
  - Planifica los pasos necesarios.
  - Interactúa con herramientas.
  - Toma decisiones en función de los resultados.

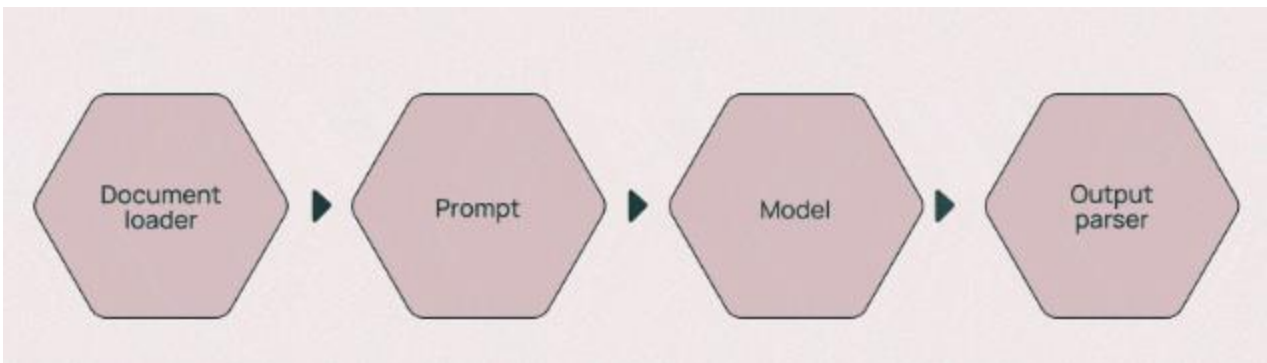
- Framework para facilitarnos el trabajo con LLMs.
- Misma interfaz para múltiples modelos.
- Permite interactuar con herramientas.
- Debemos crear un buen prompt para que funcione.
- Cuanto más específica la tarea, mejor funciona.


Tools
Output parser
Text Splitter
Prompt
Model
Chat model
Embeddings
Retriever
Document loader
Vector store


Output parser
Chat model
Tools
Prompt
Retriever
Text Splitter
Embeddings
Vector store
Model
Document loader


Retriever
Prompt
Chat model


Document loader
Text Splitter
Embeddings
Vector store


Document loader
Prompt
Model
Output parser

Limitaciones de un LLM

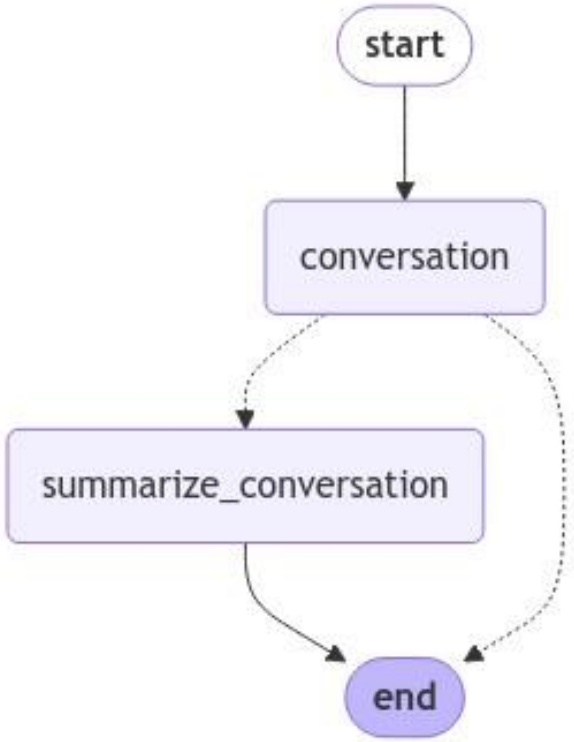
- Tenemos límite de tokens (aunque cada vez es mayor).
- Especializado en una cosa, no podemos especializarlo en otra.
- El agente no puede realizar varias tareas correctamente.
- Solo 1 prompt, solo 1 agente, solo 1 tarea.

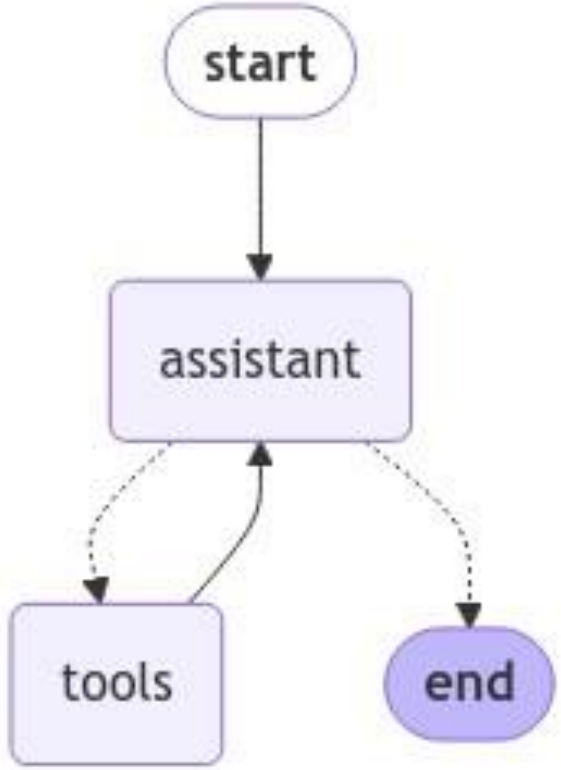
Solución: LangGraph

- Basado en nodos (grafos).
- Permite orquestar agentes de forma flexible y robusta.
- Comparten información para poder colaborar entre sí.
- Permite el uso de herramientas durante la ejecución.

Conceptos clave de grafos

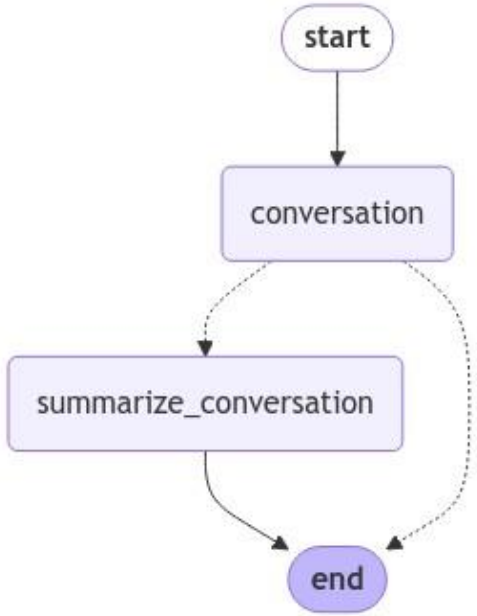
- Nodos (nodes):
  - "Puntos" donde los agentes realizan una acción.
- Aristas (edges):
  - "Lineas" que conectan nodos y definen el flujo de trabajo.
- Estado (state):
  - "Memoria" compartida que los agentes van leyendo/editando.
- Ciclos (cycles):
  - La capacidad de volver a nodos anteriores, esencial para poder iterar y refinar la solución final.





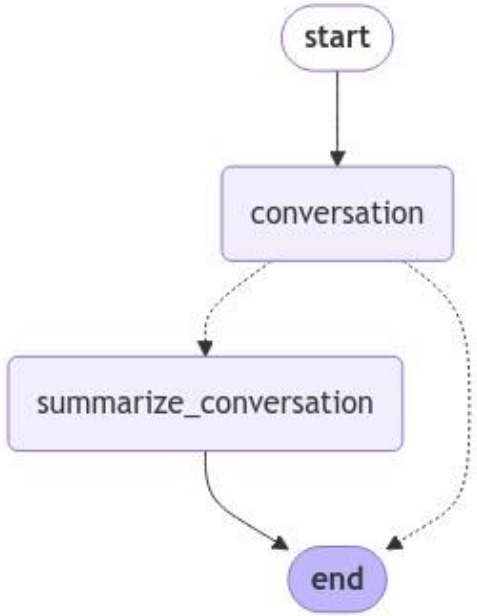
Nodos

```python
def summarize_conversation(state: State):

    # First, we get any existing summary
    summary = state.get("summary", "")

    # Create our summarization prompt
    if summary:

        # A summary already exists
        summary_message = (
            f"This is summary of the conversation to date: {summary}\n\n"
            "Extend the summary by taking into account the new messages above:"
        )

    else:
        summary_message = "Create a summary of the conversation above:"

    # Add prompt to our history
    messages = state["messages"] + [HumanMessage(content=summary_message)]
    response = model.invoke(messages)

    # Delete all but the 2 most recent messages
    delete_messages = [RemoveMessage(id=m.id) for m in state["messages"][:-2]]
    return {"summary": response.content, "messages": delete_messages}
```



Aristas (edges)

```python
# Define a new graph
workflow = StateGraph(State)
workflow.add_node("conversation", call_model)
workflow.add_node(summarize_conversation)

# Set the entrypoint as conversation
workflow.add_edge(START, "conversation")
workflow.add_conditional_edges("conversation", should_continue)
workflow.add_edge("summarize_conversation", END)

# Compile
graph = workflow.compile()
```



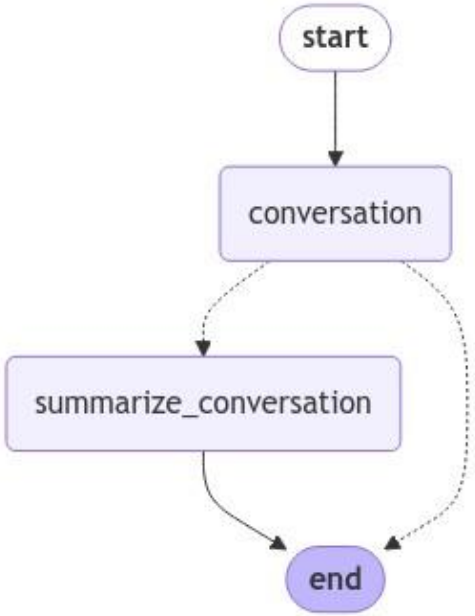

```python
def should_continue(state: State):

    """Return the next node to execute."""

    messages = state["messages"]

    # If there are more than six messages, then we summarize the conversation
    if len(messages) > 6:
        return "summarize_conversation"

    # Otherwise we can just end
    return END
```

Estado (state)

```python
from langgraph.graph import MessagesState
class State(MessagesState):
    summary: str
```

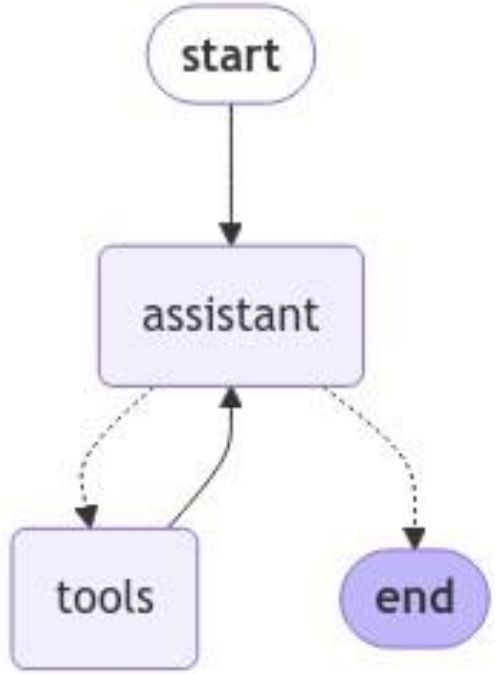
```python
class InterviewState(MessagesState):
    max_num_turns: int # Number turns of conversation
    context: Annotated[list, operator.add] # Source docs
    analyst: Analyst # Analyst asking questions
    interview: str # Interview transcript
    sections: list # Final key we duplicate in outer state for Send() API
```

```python
class TravelState(TypedDict):
    messages: Annotated[Sequence[HumanMessage | AIMessage], "Historial de mensajes"]
    destination: Annotated[str | None, "Destino del viaje"]
    start_date: Annotated[str | None, "Fecha de inicio del viaje"]
    end_date: Annotated[str | None, "Fecha de fin del viaje"]
    origin: Annotated[str | None, "Ciudad de origen"]
    flights: Annotated[list | None, "Resultados de búsqueda de vuelos"]
    accommodation: Annotated[list | None, "Resultados de búsqueda de alojamiento"]
    points_of_interest: Annotated[list | None, "Puntos de interés del destino"]
    travel_plan: Annotated[str | None, "Plan de viaje completo"]
    missing_info: Annotated[list, "Información faltante requerida"]
```



```python
from langchain_openai import ChatOpenAI

def multiply(a: int, b: int) -> int:
    """Multiply a and b.

    Args:
        a: first int
        b: second int
    """
    return a * b

# This will be a tool
def add(a: int, b: int) -> int:
    """Adds a and b.

    Args:
        a: first int
        b: second int
    """
    return a + b

def divide(a: int, b: int) -> float:
    """Divide a and b.

    Args:
        a: first int
        b: second int
    """
    return a / b

tools = [add, multiply, divide]
llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools(tools)
```

Herramientas (tools)



Ideas

- To-Do & Agenda
- Planificador de viajes
- Sistema de Research
- Ataques hacking según análisis sistema
- Simulación de toma de decisiones en emergencias
- Desarrollo de app/web/games: Design, Code, Test, Document...

Funcionalidades avanzadas

- Reducers
- Multiples Schemas
- Human-in-the-loop
- Sub-Graphs
- Time travel
- Short / Long time memory

Ejemplos:

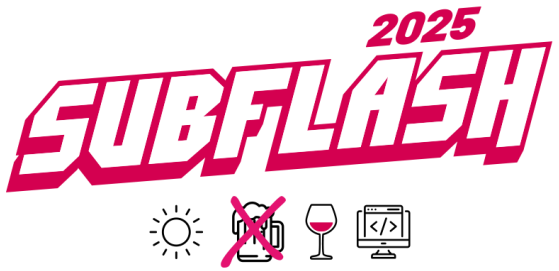
https://github.com/langchain-ai/langgraph/tree/main/examples

https://github.com/langchain-ai/langchain-academy

Otros frameworks

Crew.ai	
BeeAI IBM	
Microsoft Autogen	
Agno	

¡Gracias!



Alex Díaz
Software developer

linkedin.com/in/alex-diaz-jm
nhekoo@gmail.com