ESPECIALIZACIÓN EN DESARROLLO DE SOFTWARE
Laboratorio #1. Agentes Inteligentes

1. INTRODUCCIÓN: EL PARADIGMA DE LOS AGENTES CON ESTADO

1.1. Contexto y motivación

En el desarrollo moderno de software, los sistemas de IA ya no son simples pipelines lineales. Necesitamos agentes que:

- Mantengan contexto a través de múltiples interacciones
- Tomen decisiones basadas en estado persistente
- Manejen ciclos de razonamiento complejos
- Recuperen de errores de manera autónoma

LangGraph emerge como la solución para construir estos sistemas complejos, extendiendo las capacidades de LangChain con grafos de estado que permiten flujos de ejecución no lineales.

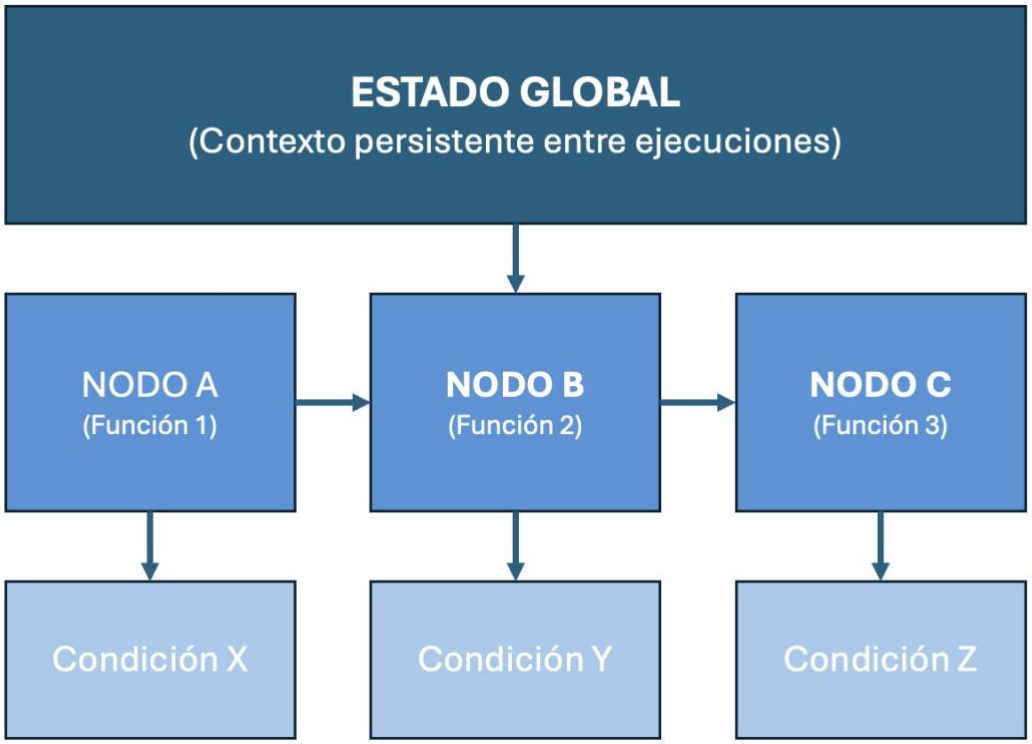
1.2. ¿Qué es LangGraph?

LangGraph es una biblioteca de Python que permite:

- Definir grafos de estado donde cada nodo representa una función o herramienta
- Gestionar transiciones condicionales entre nodos basadas en el estado actual
- Mantener estado persistente a lo largo de múltiples ciclos de ejecución
- Soportar ciclos y bucles de razonamiento (crucial para agentes reflexivos)

2. FUNDAMENTOS TEÓRICOS

2.1 Arquitectura de LangGraph



2.2 Componentes Clave

2.2.1 StateGraph

- Definición: Clase base que define la estructura del grafo

- Parámetros:
  - **state_schema:** Esquema Pydantic que define el estado
  - **config_schema:** (Opcional) Esquema para configuración
  -

2.2.2 Nodos

- Tipos:
  - Nodos regulares: Funciones que transforman el estado
  - Nodos condicionales: Funciones que deciden la siguiente ruta
  - Nodos de inicio/fin: Puntos de entrada y salida

2.2.3 Bordes (Edges)

- Tipos de transiciones:
  - Bordes regulares: graph.add_edge(origin, destination)
  - Bordes condicionales: graph.add_conditional_edges(source, condition_func, mapping)
  - Bordes por defecto: Ruta alternativa cuando no hay match

2.2.4 Checkpointers

- Propósito: Persistencia del estado entre ejecuciones

- Implementaciones:
  - MemorySaver (en memoria)
  - SQLiteSaver (base de datos SQLite)
  - PostgresSaver (base de datos PostgreSQL)

3. LABORATORIO PRÁCTICO

Guía paso a paso

1. Abrir Google Colab y crear un nuevo cuaderno.
2. Instalar las siguientes dependencias.

```
# %%capture
# Instalar paquetes necesarios (ejecutar en celda separada en Colab)
!pip install -U langchain-google-genai langgraph langchain-core langchain-community python-dotenv
```

Nota: En Google Colab, es recomendable usar %%capture para evitar salidas verbose durante la instalación, o ejecutar en celdas separadas para mejor control.

3. Ir a Google AI Studio y crear una nueva clave de API.
4. Configurar la API Key de Gemini.

```
from google.colab import userdata
import os

# Obtener API key de Google Colab Secrets
try:
    GOOGLE_API_KEY = userdata.get('GOOGLE_API_KEY')
    print("API Key de Gemini cargada correctamente")
except Exception as e:
    print("Error al cargar API Key. Configura en: Runtime > Manage secrets")
    print("Nombre del secreto: GOOGLE_API_KEY")
    raise e

# Alternativa manual (no recomendada para producción)
# os.environ["GOOGLE_API_KEY"] = "tu_api_key_aqui"
```

Seguridad: Google Colab permite almacenar credenciales de forma segura en "Secrets Manager" (en Runtime > Manage secrets), evitando exponer claves en el código.

5. Verificar los modelos disponibles en Gemini.

```
# Si quiero verificar qué modelos están disponibles:
from google.generativeai import list_models
import google.generativeai as genai

genai.configure(api_key=GOOGLE_API_KEY)
models = list_models()
print("Modelos disponibles:")
for model in models:
    print(f"- {model.name} | Métodos soportados: {model.supported_generation_methods}")
```

6. Definir esquema de estado.

```
class AgentState(TypedDict):
    messages: Annotated[Sequence[dict], operator.add]
    current_step: str
    analysis_results: dict
```

7. Inicializar LLM

```
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",  # Modelo que seleccionó en el paso 5
    temperature=0.3,
    google_api_key=GOOGLE_API_KEY
```

8. Funciones con inyección de dependencias

```
def analyze_input_gemini(state: AgentState, llm_instance):
    """Nodo que analiza la entrada usando Gemini - ¡CORREGIDO!"""
    messages = state["messages"]

    gemini_messages = [
        {"role": "system", "content": "Eres un analista experto. Analiza la solicitud del usuario y determina qué tipo de ayuda necesita."},
        *messages
    ]

    response = llm_instance.invoke(gemini_messages)
    print(f" Análisis completado: {response.content[:100]}...")  # Debug

    return {
        "current_step": "analysis_complete",
        "analysis_results": {
            "intent": "technical_support",
            "confidence": 0.92,
            "details": response.content
        }
    }

def generate_response_gemini(state: AgentState, llm_instance):
    """Nodo que genera respuesta final con Gemini - ¡CORREGIDO!"""
    messages = state["messages"]
    analysis = state["analysis_results"]

    system_prompt = f"""
    Eres un asistente técnico experto. El análisis indica:
    - Intención: {analysis['intent']}
    - Confianza: {analysis['confidence']:.2f}
    - Detalles: {analysis['details'][:200]}

    Proporciona una respuesta clara, técnica y útil. Sé conciso pero completo.
    """

    gemini_messages = [
        {"role": "system", "content": system_prompt},
        *messages
    ]

    response = llm_instance.invoke(gemini_messages)
```

```
    print(f"Respuesta generada: {response.content[:100]}...")  # Debug

    return {
        "current_step": "response_generated",
        "messages": [{"role": "assistant", "content": response.content}]
    }

```

9. Crear wrappers para el grafo.

```
def analyze_node(state):
    return analyze_input_gemini(state, llm)

def respond_node(state):
    return generate_response_gemini(state, llm)
```

10. Construir el grafo.

```
workflow = StateGraph(AgentState)
workflow.add_node("analyze", analyze_node)
workflow.add_node("respond", respond_node)
workflow.add_edge(START, "analyze")
workflow.add_edge("analyze", "respond")
workflow.add_edge("respond", END)
```

11. Compilar con checkpointing.

```
memory = MemorySaver()
app = workflow.compile(checkpointer=memory)
```

12. Ejecutar con un ejemplo real.

```
config = {"configurable": {"thread_id": "colab_test_1"}}
initial_state = {
    "messages": [
        {"role": "user", "content": "¿Cómo puedo optimizar el rendimiento de mi aplicación Flask?"}
    ],
    "current_step": "initial",
    "analysis_results": {}
}

print("Ejecutando agente con Gemini en LangGraph...")
result = app.invoke(initial_state, config=config)
```

13. Mostrar resultados

```
print("\n" + "="*50)
print("RESULTADO FINAL DEL AGENTE")
print("="*50)
print(f"Paso actual: {result['current_step']}")
print(f"Análisis: {result['analysis_results']}")
print("\n RESPUESTA DEL AGENTE:")
print(result["messages"][-1]["content"])
```

4. ACTIVIDADES A REALIZAR

AGENTE ESPECIALIZADO POR DOMINIO

Cada grupo elige un dominio técnico (DevOps, Data Science, Desarrollo Web, Seguridad, etc.) y construye un agente especializado con LangGraph que resuelva problemas específicos de ese dominio.

Entregables:

1. Documento de diseño (2-3 páginas):
   - Diagrama del grafo de estado
   - Herramienta sugerida: draw.io, Miro, o incluso diagrama a mano escaneado.
   - Debe mostrar: nodos (funciones), bordes (transiciones), y al menos una transición condicional.
   - Etiquetar claramente el estado que se actualiza en cada nodo.
   - Esquema de estado (TypedDict)
   - Incluir el código real del TypedDict usado.
   - Explicar la función de cada campo del estado (máximo 6 campos).
   - Justificar el uso de Annotated[Sequence[...], operator.add] si aplica.
   - Casos de uso y ejemplos de interacción
   - 2 escenarios concretos (no genéricos):
   - Escenario 1: Flujo normal exitoso
   - Escenario 2: Manejo de error o fallback
   - Para cada uno: mostrar entrada del usuario + salida esperada + evolución del estado.

2. Código funcional en Colab:

   - Repositorio en GitHub público con:
   - README.md que incluya: enlace al Colab, integrantes del grupo y breve descripción del dominio.
   - Historial de commits distribuido entre todos los miembros (mínimo 2 commits significativos por persona).
   - Google Colab ejecutable que contenga:
     - Al menos 3 nodos (pueden ser: 1 de análisis, 1 de procesamiento especializado, 1 condicional o de fallback).
     - Manejo de errores básico: al menos 1 bloque try/except en un nodo crítico que evite que el agente falle.
   - Checkpointing con persistencia real:
     - OPCIÓN MÍNIMA VÁLIDA: Implementación de SimpleFileSaver (guarda estado en JSON).

    - OPCIÓN RECOMENDADA: SQLiteSaver (más profesional, viable en 4 días).
- NO se acepta solo MemorySaver a menos que justifiquen en el documento cómo migrarían a persistencia real.
- Código limpio y comentado: funciones con docstrings, nombres significativos, sin código comentado innecesario.

Importante: El Colab debe ejecutarse completo de principio a fin sin errores en condiciones normales.

3. Demostración en video (máximo 5 minutos):
    - Presentar al grupo y el dominio técnico elegido.
    - Mostrar el diagrama del grafo de estado (pueden usar draw.io o Miro).
    - Mostrar 3 escenarios distintos, uno tras otro, en vivo en Colab.
    - Ejecutar una consulta típica del dominio. Mostrar cómo el estado se actualiza y se mantiene entre interacciones (ej: "Recordar la última pregunta").
    - Ingresar una entrada inválida o que cause un error (ej: código mal formado, pregunta sin sentido). Mostrar cómo el agente no se rompe y responde con un mensaje amigable o fallback.
    - Mostrar cómo el agente cambia de ruta basado en el estado (ej: si la consulta es compleja, activa un nodo especializado; si es simple, responde directamente).

5. CRITEROS DE EVALUACIÓN

Criterio de evaluación	5.0 (Excelente)	4.0 (Satisfactorio)	3.0 (Aceptable)	2.0 o menos (Insuficiente)
1. Arquitectura del agente (25%) Claridad del diseño, diagrama, esquema de estado	• Diagrama completo con nodos, transiciones y condicionales. • `TypedDict` bien estructurado (≤6 campos). • Casos de uso realistas y bien explicados.	• Diagrama claro pero simple. • Esquema de estado funcional. • 2 casos de uso descritos.	• Diagrama básico (puede ser a mano). • Esquema de estado mínimo. • 1 caso de uso válido.	• Diagrama ausente, incompleto o incoherente. • Estado mal definido o no usado.

2. Implementación técnica (35%) Código funcional, nodos, manejo de errores, persistencia	• ≥3 nodos funcionales (incluye condicional). • Persistencia real (SQLite/JSON) funcionando. • Manejo de errores robusto en ≥2 nodos. • Código limpio y bien organizado.	• 3 nodos básicos. • Persistencia real implementada (aunque simple). • 1 manejo de errores funcional. • Código legible.	• 2 nodos funcionales. • Usa `MemorySaver` pero explica cómo migraría a persistencia real en el documento. • 1 try/except básico. • Código ejecutable sin errores.	• Menos de 2 nodos. • No hay persistencia ni explicación. • El Colab no se ejecuta o falla en flujo básico.
3. Demostración en video (25%) Claridad, cumplimiento de los 3 bullets, evidencia del estado y errores	• Video ≤5 min, bien estructurado. • Muestra claramente los 3 escenarios.• Explica y demuestra persistencia de estado. • Muestra recuperación real ante error.	• Video claro, dentro del tiempo. • Muestra 3 escenarios (1 puede ser débil). • Menciona persistencia y errores.	• Video ≤5 min. • Muestra 2 escenarios (flujo normal + error). • Menciona persistencia (aunque no la muestre en detalle).	• Video ausente, >5 min, o sin demostración real. • No muestra persistencia ni manejo de errores.
4. Trabajo en equipo y entrega (15%) GitHub, distribución de tareas, calidad de entrega	• Repositorio GitHub con buen historial (commits distribuidos). • README claro, enlace al Colab. • Entrega completa y profesional.	• GitHub con actividad de todos. • Entrega completa, aunque básica.	• GitHub con mínimo 2 commits por persona. • Entrega justa a tiempo.	• Repositorio vacío, sin actividad grupal, o entrega incompleta.