# CURSO COMPLETO DE LANGCHAIN, LANGGRAPH Y AGENTES DE IA CON PYTHON

SANTIAGO HERNÁNDEZ RAMOS

# Tema 1 – Introducción a LangChain: Tu primera aplicación con IA y LLMs

## Guía de referencia

**Usa esta guía como recordatorio rápido de componentes, clases y parámetros clave.** Está pensada para cualquier proyecto, no solo para los ejemplos del curso.

---

### 1 · Configuración del entorno

| Componente | Comando/Acción | Comentario |
|---|---|---|
| **Entorno virtual** | `python -m venv venv` | Crea un entorno aislado para las dependencias del proyecto |
| **Activar entorno** | `venv\Scripts\activate` (Windows) | Verifica que aparezca (venv) en tu terminal |
| **Instalar LangChain** | `pip install langchain langchain-openai langchain-google-genai` | Instala el framework base y conectores para proveedores |
| **Estructura de carpetas** | Organizar por temas/módulos | Facilita la navegación y mantenimiento del código |

---

### 2 · Proveedores y modelos (clases y opciones)

| Proveedor | Clase LangChain | Parámetros clave (ejemplos) | Notas |
|---|---|---|---|
| OpenAI | `from langchain_openai import ChatOpenAI` | `model="gpt-4-mini"`, `temperature=0.0–1.0` | `temperature` ↑ → más creatividad/aleatoriedad; ↓ → más determinismo/consistencia. |
| Google GenAI (Gemini) | `from langchain_google_genai import ChatGoogleGenerativeAI` | `model="gemini-2.5-flash"`, `temperature=…` | Interfaz equivalente: cambiar proveedor requiere mínimos cambios. |

**Patrón común de inicialización**

```
llm = ChatOpenAI(model="gpt-4-mini", temperature=0.7)
# o
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.7)
```

---

### 3 · Gestión de credenciales (variables de entorno)

| Sistema | Variable típica | Ejemplo de definición | Uso en código |
|---|---|---|---|
| Windows (CMD) | `OPENAI_API_KEY` | `setx OPENAI_API_KEY "tu_clave"` | LangChain las detecta automáticamente al crear `ChatOpenAI`. |
| Windows (CMD) | `GOOGLE_API_KEY` | `setx GOOGLE_API_KEY "tu_clave"` | Reabre VS Code/terminal para recargar el entorno. |
| macOS/Linux (bash/zsh) | `OPENAI_API_KEY` / `GOOGLE_API_KEY` | `export OPENAI_API_KEY="tu_clave"` | Añádelo a tu `~/.bashrc` / `~/.zshrc` para persistir. |

**Buenas prácticas**

- Nunca hardcodees la API key en el script.
- Cierra y reabre la terminal tras definir la variable para que el proceso herede el valor.

---

## 4 · Invocación básica del LLM (Hello World)

| Paso | Qué haces | Snippet |
|---|---|---|
| Definir prompt | Texto de entrada (puede venir de usuario o del programa) | `pregunta = "¿En qué año llegó el ser humano a la Luna por primera vez?"` |
| Invocar | Usas la interfaz unificada de LangChain | `respuesta = llm.invoke(pregunta)` |
| Leer resultado | Obtienes el contenido generado | `print(respuesta.content)` |

Notas

- `.invoke()` es la interfaz estándar (sin importar el proveedor).
- El objeto devuelto suele incluir `content` con el texto generado.

---

## 5 · Prompt Templates (plantillas reutilizables y con variables)

| Componente | Función | Sintaxis |
|---|---|---|
| **PromptTemplate** | Define prompts reutilizables con variables | `from langchain.prompts import PromptTemplate` |
| **input_variables** | Lista de variables dinámicas | `input_variables=["nombre", "tema"]` |
| **template** | Texto con placeholders | `"Hola {nombre}, hablemos de {tema}"` |
| **Variables** | Se sustituyen en tiempo de ejecución | Usar `{variable}` dentro del template |

Patrón de uso

```
from langchain.prompts import PromptTemplate

plantilla = PromptTemplate(
    input_variables=["nombre"],
    template="Saluda al usuario.\nNombre del usuario: {nombre}\nAsistente:"
)

# (Opcional) Render manual:
prompt_renderizado = plantilla.format(nombre="Carlos")
```

Consejos

- Usa `\n` para estructurar instrucciones (roles, pasos, secciones).
- Declara todas las variables dinámicas en `input_variables` .
- Mantén el prompt claro y específico (instrucciones → contexto → salida esperada).

---

## 6 · Cadenas (Chains) con LangChain

Objetivo: Encadenar pasos como “plantilla → LLM → salida”, con una API coherente.

Enfoque clásico (deprecado pero aún visto en código existente):

```
from langchain.chains import LLMChain

chain = LLMChain(llm=llm, prompt=plantilla)
resultado = chain.run(nombre="Carlos")  # pasa las variables del prompt
print(resultado)  # Texto generado por el LLM
```

| Punto | Clave |
|---|---|
| `LLMChain` | API clásica; verás avisos de *deprecated*. Aún útil para leer/entender proyectos existentes. |
| Invocación | `.run(**kwargs)` pasa las variables para sustituir en la plantilla. |
| Evolución | La tendencia actual es usar la sintaxis basada en *runnables/LCEL* (se cubrirá en secciones posteriores). |

---

## 7 · Parámetros y prácticas recomendadas

- Temperatura:
  - `0.0–0.2` → Respuestas precisas/estables (QA factual, extracción).
  - `0.3–0.7` → Equilibrio general (asistentes, reformulación).
  - `0.8–1.0` → Creatividad alta (brainstorming, copy).
- Trazabilidad: imprime/guarda `prompt` y parámetros (útil para depuración).
- Versiona prompts: tratar plantillas como artefactos (pueden romper resultados).
- Estructura por temas: carpetas por módulo/lección facilitan localizar ejemplos.
- Proveedor intercambiable: misma interfaz ( `.invoke` ) para OpenAI/Gemini → reduce *vendor lock-in*.

---

## 8 · LCEL (LangChain Expression Language) — cadenas modernas

| Concepto | Cómo se hace con LCEL | Ventaja |
|---|---|---|
| Construir cadena | `pipeline = plantilla \| llm` | Interfaz estándar y consistente |
| Invocar | `result = pipeline.invoke({"var": "valor"})` | Interfaz unificada con `.invoke()` (igual que un LLM). |
| Paso de variables | Diccionario con claves = `input_variables` | Más claro que kwargs dinámicos. |
| Salida | `result.content` (objeto `BaseMessage` ) | Estandariza la respuesta con LLMs. |
| Sustituye a | `LLMChain(...).run(...)` (deprecado) | Evita APIs distintas, warnings y ambigüedad. |

Patrón mínimo

```
from langchain.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4-mini", temperature=0.5)
tpl = PromptTemplate(input_variables=["nombre"],
                     template="Saluda a {nombre} como si fueras un asistente cortés.")
chain = tpl | llm
msg = chain.invoke({"nombre": "Carlos"})   # dict, no kwargs sueltos
print(msg.content)
```

---

## 9 · Tipos de mensajes y roles en LangChain

| Clase | Rol | Uso típico |
|---|---|---|
| **SystemMessage** | Sistema/Desarrollador | Instrucciones de comportamiento, tono, límites. (No se muestran al usuario.) |
| **HumanMessage** | Usuario | Preguntas y peticiones del usuario |
| **AIMessage** | Asistente | Respuestas del modelo LLM |
| **BaseMessage** | Clase padre | Base de todos los mensajes |

```python
from langchain.schema import AIMessage, HumanMessage, SystemMessage

# Ejemplo de uso
mensajes = [
    SystemMessage(content="Eres un asistente útil"),
    HumanMessage(content="¿Cuál es la capital de Francia?"),
    AIMessage(content="La capital de Francia es París")
]
```

---

## 10 · Streamlit para aplicaciones IA

| Componente | Método | Función |
|---|---|---|
| **Configuración página** | `st.set_page_config()` | Título, favicon, layout |
| **Elementos UI** | `st.title()` , `st.markdown()` | Añadir títulos y texto |
| **Session State** | `st.session_state` | Persistir datos entre ejecuciones |
| **Chat UI** | `st.chat_message(role)` | Mostrar mensajes de chat |
| **Input** | `st.chat_input()` | Campo de entrada para usuario |
| **Ejecutar app** | `streamlit run archivo.py` | Lanza servidor local |

```python
import streamlit as st

# Configuración inicial
st.set_page_config(
    page_title="Mi Chatbot",
    page_icon="🤖"
)

# Inicializar estado
if "mensajes" not in st.session_state:
    st.session_state.mensajes = []

# Mostrar mensaje con rol
with st.chat_message("assistant"):
    st.markdown("Hola, ¿en qué puedo ayudarte?")
```

---

## 11 · Gestión de memoria en chatbots

| Estrategia | Implementación | Consideración |
|---|---|---|
| **Session State** | `st.session_state["mensajes"]` | Persiste durante la sesión del navegador |
| **Lista de mensajes** | Almacenar como lista Python | Fácil de iterar y actualizar |
| **Filtrado de roles** | `isinstance(msg, SystemMessage)` | No mostrar mensajes del sistema |
| **Historial completo** | Enviar todos los mensajes al LLM | Mantiene contexto de conversación |

```python
# Patrón para gestión de memoria
if "mensajes" not in st.session_state:
    st.session_state.mensajes = []

# Mostrar historial
for mensaje in st.session_state.mensajes:
    if isinstance(mensaje, SystemMessage):
        continue  # No mostrar mensajes del sistema

    role = "assistant" if isinstance(mensaje, AIMessage) else "user"
    with st.chat_message(role):
        st.markdown(mensaje.content)
```

---

## 12 · Generación de respuestas con contexto

| Estrategia | Método | Ventajas |
|---|---|---|
| **Invoke directo** | `llm.invoke(st.session_state.mensajes)` | Simple pero sin formato específico |
| **Con plantilla** | `prompt \| llm` con LCEL | Control sobre comportamiento y formato |
| **Streaming** | `.stream()` en lugar de `.invoke()` | Respuesta progresiva como ChatGPT |
| **Variables dinámicas** | `{"mensaje": pregunta, "historial": mensajes}` | Contexto estructurado |

```python
# Con plantilla de prompt (recomendado)
prompt = PromptTemplate(
    input_variables=["historial", "mensaje"],
    template="""Eres un asistente útil y amigable.
    Historial: {historial}
    Responde a: {mensaje}
    Asistente:"""
)

cadena = prompt | llm
respuesta = cadena.invoke({
    "historial": st.session_state.mensajes,
    "mensaje": pregunta
})
```

---

## 13 · Respuestas en streaming

| Paso | Código | Propósito |
|---|---|---|
| **Stream chunks** | `for chunk in chain.stream(...)` | Recibir fragmentos |
| **Concatenar** | `full_response += chunk.content` | Construir respuesta completa |
| **Actualizar UI** | `placeholder.markdown(full_response + "▌")` | Mostrar progreso con cursor |
| **Finalizar** | `placeholder.markdown(full_response)` | Mostrar sin cursor al terminar |

```python
# Patrón completo de streaming
response_placeholder = st.empty()
full_response = ""

for chunk in cadena.stream({"mensaje": pregunta,
                            "historial": st.session_state.mensajes}):
    full_response += chunk.content
    response_placeholder.markdown(full_response + "▌")

# Mostrar respuesta final sin cursor
response_placeholder.markdown(full_response)

# Guardar en memoria
```

```
st.session_state.mensajes.append(
    AIMessage(content=full_response)
)
```

# Página de Notas del Tema

Esta página está pensada para que puedas anotar ideas clave, dudas y reflexiones importantes sobre el tema anterior.

# Tema 2 – Fundamentos de LangChain: Componentes Core

## Guía de referencia

**Usa esta guía como recordatorio rápido de los componentes y APIs clave.** Está pensada para cualquier proyecto, no solo para los ejemplos del curso.

---

### 1 · Runnables (núcleo de las "cadenas")

| Concepto | ¿Qué es? | ¿Cuándo usarlo? | API/Operaciones clave | Notas |
|---|---|---|---|---|
| **Runnable** | Cualquier objeto invocable con `.invoke(input) → output`. | Para encapsular pasos de procesamiento (pre/post), modelos, parsers, etc. | `.invoke(x)` | Base de todo el LangChain Expression Language (LCEL). |
| **Pipe ( `\|` )** | Operador de composición secuencial. | Para encadenar pasos donde la salida de uno es la entrada del siguiente. | `r1 \| r2 \| r3` | |
| `RunnableLambda` | Adapta una función Python (lambda/def) a Runnable. | Para integrar lógica propia (limpieza, formateo, validaciones). | `RunnableLambda(fn)` | Permite mezclar componentes LangChain con utilidades Python propias. |

**Patrón típico**

1. Envuelve funciones Python con `RunnableLambda` (p. ej., formatear/transformar).
2. Conecta con otros runnables (modelo, parser) usando `|`.
3. Ejecuta con `.invoke(input)` y recibe el resultado del último eslabón.

**Buenas prácticas**

- Asigna nombres descriptivos a cada paso (mejor trazabilidad).
- Aísla transformaciones de datos en funciones pequeñas y puras.
- Piensa en tipos de entrada/salida por paso (strings, dicts, listas).

---

### 2 · Paralelización con `RunnableParallel`

| Objetivo | ¿Cuándo usarlo? | ¿Qué hace? | API/Operaciones clave | Salida |
|---|---|---|---|---|
| Ejecutar ramas independientes a la vez | Cuando varios pasos consumen **el mismo input** y no dependen entre sí (p. ej., **resumir** y **analizar sentimiento** sobre el mismo texto preprocesado). | "Abre" el flujo en ramas concurrentes y devuelve todos los resultados juntos. | `RunnableParallel({key: runnable, ...})` | `dict` con las claves definidas (p. ej., `{"resumen": "...", "sentiment_data": {...}}`). |

**Cómo encaja en la cadena**

```
preprocesamiento | RunnableParallel({ "resumen": summaryBranch, "sentiment_data": sentimentBranch }) | mergeResults
```

**Notas y cuidados**

- El input de `RunnableParallel` se **reutiliza** para cada rama.
- Las ramas no garantizan orden; diseña tu merge para claves, no posiciones.
- Úsalo para ahorrar tiempo cuando el modelo/recursos lo permitan.

---

## 3 · Procesamiento por lotes con `.batch(...)`

| Problema | Solución | Beneficios | API/Operaciones clave | Salida |
|---|---|---|---|---|
| Procesar muchas entradas (reseñas, consultas, etc.) de forma eficiente. | Ejecuta la **misma cadena** sobre una **lista** de inputs simultáneamente. | Mejor uso de recursos; menos sobrecarga; **gestión de errores por elemento** (si uno falla, continúan los demás); código más simple. | `chain.batch([in1, in2, in3, ...])` | `list` de resultados en el **mismo orden** de entrada. |

**Cuándo preferir `.batch` sobre un bucle con `.invoke`**

- Siempre que tengas 2+ elementos a procesar y no requieras control personalizado paso a paso.
- Especialmente útil con modelos remotos: reduce latencias acumuladas.

**Consejos**

- Mantén cada unidad de trabajo "atómica" (una reseña, una consulta, etc.).
- Diseña la cadena para devolver objetos estructurados (p. ej., dict/JSON) y facilitar el post-proceso.

---

## 4 · Prompt Templates y Prompt Engineering

| Concepto | ¿Qué es? | ¿Cuándo usarlo? | API/Operaciones clave | Buenas prácticas |
|---|---|---|---|---|
| `PromptTemplate` | Plantilla de texto con partes fijas (instrucciones/contexto) y variables (placeholders). | Cuando quieras **separar** instrucciones del contenido dinámico (preguntas, datos). | `PromptTemplate(template=..., input_variables=[...])` · `prompt.format(**vars)` | Define la **plantilla** en una variable aparte y luego crea el objeto. Nombra claramente cada `input_variable`. |
| **Validación local** | Probar cómo se "rellena" el prompt sin llamar al LLM. | Antes de integrar en la cadena, para verificar posiciones y sustituciones. | `prompt.format(...)` | Ahorra tiempo y errores; ideal cuando mezclas mucho texto e inputs. |
| **Prompt engineering** | Disciplina para **diseñar** prompts efectivos (claridad, rol, formato de respuesta, restricciones). | Siempre: el prompt condiciona por completo el comportamiento del LLM. | — | Itera, testea y versiona plantillas; sé explícito (rol, estilo, longitud, formato de salida). |

**Patrón recomendado**

1. Declara `template` (texto con placeholders).
2. Crea `PromptTemplate(template=template, input_variables=[...])`.
3. Valida con `prompt.format(...)`.
4. Úsalo dentro de tu cadena (con el modelo, parsers, etc.).

---

## 5 · ChatPromptTemplate (plantillas para chats)

| Concepto | ¿Qué es? | ¿Cuándo usarlo? | API/Operaciones clave | Notas |
|---|---|---|---|---|
| `ChatPromptTemplate` | Plantilla de **lista de mensajes** (con roles) para LLMs conversacionales. | Cuando separas instrucciones de **sistema** y contenido del **usuario** (y opcionalmente del **asistente**). | `ChatPromptTemplate.from_messages([...])` · `prompt.format_messages(**vars)` | Es la **opción por defecto** para flujos conversacionales. |
| **Roles** | `system` , `human` (o `user` ), `ai` (o `assistant` ). | Definir comportamiento, entrada del usuario y referencias de respuestas previas. | Mensajes como tuplas: `("system", "...")` , `("human", "{texto}")` | Compatible con modelos tipo chat; evita "concatenar" texto a mano. |
| **Variables** | Placeholders en cualquier mensaje. | Para inyectar datos dinámicos en mensajes concretos. | `{nombre_variable}` | Valídalo con `format_messages` antes de invocar al modelo. |

**Patrón recomendado**

```
prompt = ChatPromptTemplate.from_messages([
  ("system", "Actúa como traductor ES→EN, sé preciso."),
  ("human", "{texto}")
])
messages = prompt.format_messages(texto="Hola, ¿cómo estás?")
# En cadena LCEL:
# chain = prompt | chat_model | StrOutputParser()
```

**Consejos**

- Mantén el **mensaje de sistema** claro (rol, objetivo, estilo, límites).
- Usa variables con nombres expresivos ( `{consulta}` , `{contexto}` , `{idioma_destino}` ).

---

## 6 · `MessagesPlaceholder` (inyectar historiales y listas de mensajes)

| Problema | Solución | ¿Qué permite? | API/Operaciones clave | Entrada esperada |
|---|---|---|---|---|
| Necesitas insertar **múltiples mensajes** (historial, few-shots) en la plantilla. | `MessagesPlaceholder` dentro de `ChatPromptTemplate` . | Reutilizar un **listado de mensajes** ya tipados ( `HumanMessage` , `AIMessage` , etc.). | `MessagesPlaceholder(variable_name="historial")` · `prompt.format_messages(historial=[...], pregunta_actual="...")` | Lista de `BaseMessage` (no un string concatenado). |

**Patrón típico con contexto + nueva pregunta**

```
prompt = ChatPromptTemplate.from_messages([
  ("system", "Eres un asistente útil y mantienes el contexto."),
  MessagesPlaceholder(variable_name="historial"),
  ("human", "{pregunta_actual}")
])
# luego:
messages = prompt.format_messages(
  historial=historial_conversacion,  # lista de mensajes
  pregunta_actual="¿Puedes detallar la arquitectura?"
)
```

**Cuándo usarlo**

- Chats con **memoria** (persistir turnos anteriores).

- **In-Context Learning / Few-shot** (inyectar ejemplos como mensajes).

**Buenas prácticas**

- Controla el **límite de tokens** recortando historial (ventana deslizante, resúmenes).
- Etiqueta claramente los mensajes de ejemplo (quién pregunta/responde).

---

## 7 · Plantillas por rol reutilizables

| Concepto | ¿Qué es? | ¿Cuándo usarlo? | API/Operaciones clave | Ventaja |
|---|---|---|---|---|
| `SystemMessagePromptTemplate` | Plantilla **solo** para rol de sistema. | Reutilizar instrucciones de sistema con variables (rol, tono, normas). | `SystemMessagePromptTemplate.from_template("... {tono}")` | Modularidad; evita duplicar texto. |
| `HumanMessagePromptTemplate` | Plantilla **solo** para rol humano/usuario. | Normalizar cómo formateas la entrada de usuario en distintos módulos. | `HumanMessagePromptTemplate.from_template("Mi pregunta sobre {tema} es: {pregunta}")` | Homogeneiza la UX del prompt. |
| **Composición** | Combinar varias plantillas por rol en un prompt de chat. | Cuando distintos componentes aportan mensajes predefinidos. | `ChatPromptTemplate.from_messages([system_t, human_t])` | Jerarquía y reutilización limpias. |

**Patrón recomendado (composición)**

```
system_t = SystemMessagePromptTemplate.from_template(
  "Eres un {rol} especializado en {especialidad}. Responde con tono {tono}."
)
human_t  = HumanMessagePromptTemplate.from_template(
  "Mi pregunta sobre {tema} es: {pregunta}"
)
chat_prompt = ChatPromptTemplate.from_messages([system_t, human_t])
messages = chat_prompt.format_messages(
  rol="nutricionista", especialidad="dietas veganas",
  tono="profesional pero accesible",
  tema="proteínas vegetales",
  pregunta="¿Mejores fuentes para un atleta?"
)
```

**Tips**

- Centraliza tus plantillas por rol en **módulos** reutilizables.
- Versiona cambios de plantillas (p. ej., `system_v1` , `system_v2` ) para A/B testing.

---

## 8 · Output Parsers (estructurar la salida del LLM)

| Objetivo | ¿Por qué es clave? | Parsers comunes | Encaje en LCEL | Salida |
|---|---|---|---|---|
| Forzar/convertir la salida a un **formato útil**. | La mayoría de apps **no** consumen texto libre; necesitan JSON/CSV/objetos. | `JsonOutputParser` , `PydanticOutputParser` , `StrOutputParser` , `CsvOutputParser` | Se conectan al final de la cadena: `prompt \| model \| parser` | Tipos nativos (dict/list), strings o instancias Pydantic. |

**JSON "estricto"**

- Usa `JsonOutputParser()` cuando esperes un **JSON válido**.
- Añade **instrucciones de formato** al prompt (si el parser las expone) o valida post-salida.

### Modelos de datos con `PydanticOutputParser`

- Define un **modelo** Pydantic con campos y tipos.
- El parser **valida y castea** la salida al modelo (lanza error si no encaja).
- Útil para **contratos** entre componentes (APIs internas, pipelines).

#### Patrón con Pydantic

```
from pydantic import BaseModel, Field
from langchain_core.output_parsers import PydanticOutputParser

class Analisis(BaseModel):
    resumen: str = Field(..., description="Resumen en una frase")
    sentimiento: str = Field(..., description="positivo|negativo|neutro")
    razon: str

parser = PydanticOutputParser(pydantic_object=Analisis)

# (Opcional) Instrucciones de formato para el prompt:
# format_instructions = parser.get_format_instructions()
# ...inclúyelas en el mensaje de sistema/usuario.

chain = prompt | chat_model | parser
resultado: Analisis = chain.invoke(entrada)
```

#### Buenas prácticas

- **Siempre** termina tu cadena con un parser (aunque sea `StrOutputParser`).
- Si el parser ofrece `get_format_instructions()`, **inyéctalas** en el prompt.
- Maneja **errores de parseo** (reintentos con `with_structured_output`/fallbacks).
- Mantén los esquemas **mínimos y estables** (evita campos ambiguos).

---

## 9 · Pydantic (modelado y validación de datos)

| Concepto | ¿Para qué sirve? | ¿Cómo se usa? | Ventajas |
|---|---|---|---|
| `BaseModel` | Definir **esquemas** de datos con tipos y validación. | `class Usuario(BaseModel): id:int; nombre:str; activo:bool=True` | Convierte tipos (p. ej. `"123"→123`), valores por defecto, validación. |
| `Field` | Añadir metadatos/validaciones. | `edad:int = Field(..., ge=0, le=120, description="Años")` | Reglas de rango, descripciones para guiar al LLM. |
| **Serialización** | Exportar modelos a formatos comunes. | `.model_dump()`, `.model_dump_json()` | Integración sencilla con APIs/BDs. |

#### Patrón básico

```
from pydantic import BaseModel, Field

class Usuario(BaseModel):
    id: int
    nombre: str
    activo: bool = True

u = Usuario(id="123", nombre="Ana")  # convierte tipos
json_str = u.model_dump_json()
```

#### Buenas prácticas

- Tipa **todo** (incluye listas, dicts y submodelos).
- Usa `description` en `Field` para que los LLMs entiendan qué devolver.
- Restringe con `ge`, `le`, `min_length`, etc., para salidas más fiables.

---

## 10 · Salidas estructuradas con LangChain + Pydantic

| Objetivo | ¿Qué hace? | API clave | Dónde encaja |
|---|---|---|---|
| Forzar al LLM a devolver **objetos tipados** | El modelo responde en el **esquema Pydantic** indicado. | `chat_model.with_structured_output(MyModel)` → `.invoke(...)` | Al final de la cadena LCEL: `prompt \| chat \| (structured)` . |

**Patrón recomendado**

```python
from pydantic import BaseModel, Field

class Analisis(BaseModel):
    resumen: str = Field(..., description="Resumen en una frase")
    sentimiento: str = Field(..., description="positivo|neutro|negativo")

chat = ChatOpenAI(model="gpt-4o-mini", temperature=0.6)
structured = chat.with_structured_output(Analisis)

resultado: Analisis = structured.invoke("Analiza: ...")
print(resultado.sentimiento)          # acceso tipado
print(resultado.model_dump_json())    # JSON válido
```

**Consejos**

- Incluye **descripciones** en todos los campos; el LLM las usa como contrato.
- Si necesitas números/arrays anidados, tipa explícitamente ( `int` , `list[str]` , submodelos).
- Maneja **errores de parseo** (reintentos/fallback a `StrOutputParser` ).

---

## 11 · Arquitectura de proyectos LLM (estructura sugerida)

| Carpeta | Contenido típico | Por qué |
|---|---|---|
| `models/` | **Modelos Pydantic** (esquemas de salida/entrada). | Contratos estables entre módulos. |
| `prompts/` | **ChatPromptTemplate**, plantillas por rol, few-shots. | Reutilización y versionado de prompts. |
| `services/` | Utilidades: E/S de ficheros, parsers, conectores API. | Aísla dependencias externas. |
| `ui/` | Interfaz (CLI/Web: Streamlit/Gradio/FastAPI). | Separa presentación de lógica. |

**Buenas prácticas**

- Empieza por el **modelo de salida** (qué esperas del LLM) y diseña hacia atrás.
- Versiona prompts ( `system_v1` , `system_v2` ) y **testea** con datos sintéticos.
- Mantén cadenas LCEL **pequeñas y componibles**.

---

## 12 · Diseñar esquemas Pydantic eficaces para LLMs

| Elemento | Recomendación | Ejemplo |
|---|---|---|
| Campos string | Define propósito en `description` . | `nombre:str = Field(...,"Nombre completo extraído del documento")` |
| Números | Restringe rango y unidad. | `ajuste:int = Field(..., ge=0, le=100, description="0-100")` |
| Listas | Limita tamaño esperado (descripción) y tipo interno. | `habilidades:list[str] = Field(...,"Top 5-7 habilidades")` |
| Submodelos | Representa bloques (experiencia, educación, etc.). | `class Experiencia(BaseModel): periodo:str; rol:str` |

**Plantilla general**

```python
class Resultado(BaseModel):
    titulo: str = Field(..., description="Título conciso")
    items: list[str] = Field(..., description="Lista de puntos clave (3–5)")
    score: int = Field(..., ge=0, le=100, description="Puntuación global")
```

---

## 13 · Ingesta de PDFs: extracción de texto (Python)

| Tarea | Herramientas | API/esqueleto | Tips |
|---|---|---|---|
| Leer PDF en memoria | `PyPDF2` , `io.BytesIO` | `PdfReader(BytesIO(pdf_bytes))` | Acepta **bytes** desde UI/web. |
| Extraer texto por página | `page.extract_text()` | Itera `for i, page in enumerate(reader.pages): ...` | Añade separadores entre páginas. |
| Limpieza | `str.strip()` / normalización | Quita líneas en blanco y espacios extra. | **Ahorra tokens** y coste. |
| Errores | `try/except` | Devuelve mensaje claro en errores. | PDFs escaneados → OCR externo. |

**Patrón básico**

```python
from io import BytesIO
from PyPDF2 import PdfReader

def extraer_texto_pdf(pdf_bytes: bytes) -> str:
    try:
        reader = PdfReader(BytesIO(pdf_bytes))
        partes = []
        for i, page in enumerate(reader.pages, start=1):
            txt = (page.extract_text() or "").strip()
            if txt:
                partes.append(f"\n--- Página {i} ---\n{txt}\n")
        texto = "".join(partes).strip()
        if not texto:
            return "ERROR: PDF vacío o solo imágenes."
        return texto
    except Exception as e:
        return f"ERROR: al procesar el PDF: {e}"
```

**Buenas prácticas**

- Para PDFs **escaneados** usa OCR (p. ej., Tesseract/DocTR) antes de pasar al LLM.
- Recorta texto si supera el **límite de tokens** (resúmenes por página o chunking).
- Normaliza codificación, bullets y tablas (considera parsers específicos si necesitas estructura).

# Página de Notas del Tema

Esta página está pensada para que puedas anotar ideas clave, dudas y reflexiones importantes sobre el tema anterior.

# Tema 3 – RAG y LangChain: Cargando y recuperando datos del mundo real

## Guía de referencia

**Usa esta guía como recordatorio rápido de componentes, opciones y "pitfalls"**. Está pensada para cualquier proyecto con LangChain, no solo para el ejemplo del curso.

---

### 1 · LangChain Community (integraciones externas)

**Qué es:** Paquete separado que concentra *integraciones de terceros* (loaders, stores, conectores a APIs y BBDD).
**Relación con `langchain-core`:** Core = abstracciones y protocolos; Community = *implementaciones* concretas.

**Instalación básica**

- Entorno: activa tu venv.
- Paquete: `pip install langchain-community`
- Imports típicos: `from langchain_community.document_loaders import …`

**Cuándo usarlo**

- Cargar/procesar documentos (PDF, web, GDrive, Notion, Slack…).
- Conectarte a bases vectoriales (FAISS, Chroma, Pinecone, Milvus, Qdrant…).
- Integrarte con APIs de terceros (Google, etc.).

**Patrón de dependencias**

- Muchas integraciones requieren librerías extra (p. ej., `pypdf`, `beautifulsoup4`, SDKs de proveedores).
- Error típico: *ModuleNotFoundError* → instala la dependencia indicada por el mensaje.

**Buenas prácticas**

- Fijar versiones en `requirements.txt`.
- Aislar secretos en variables de entorno/gestores (no subir `credentials.json` a VCS).
- Crear un módulo `adapters/` donde centralizas loaders/conectores para cambiar proveedor sin tocar el resto del código.

---

### 2 · Document Loaders (cargar datos en objetos `Document`)

**Qué hacen:** Normalizan entradas externas a objetos `Document` con:

- `page_content`: texto (o contenido) del recurso.
- `metadata`: diccionario con contexto (ruta, url, título, autor, página, timestamps, etc.).

**Flujo típico**

1. Instanciar loader → 2) `load()` (o `lazy_load()`) → 3) lista/iterable de `Document` → 4) procesar/splitear/embeddings.

**Loaders comunes y notas**

| Loader | Uso típico | Dependencias frecuentes | Notas clave |
|---|---|---|---|
| `PyPDFLoader` | Leer PDFs, 1 `Document` por página | `pypdf` | Devuelve **lista** (páginas). `metadata` incluye número de página. |
| `WebBaseLoader` | Extraer HTML → texto | `beautifulsoup4` | Webs dinámicas pueden requerir alternativas (Selenium/Playwright). |
| `CSVLoader`, `Unstructured*` | Archivos ofimáticos variados | `unstructured`, `pandas` | `unstructured` maneja docx, pptx, etc. |
| `NotionDBLoader`, `SlackDirectoryLoader`, `GmailLoader`, etc. | SaaS | SDK/credenciales del proveedor | Requiere OAuth/keys y scopes adecuados. |
| Directorios (`DirectoryLoader`) | Carpeta con múltiples ficheros | según subloader | Combínalo con *glob* y subloaders específicos. |

**API de uso**

```python
from langchain_community.document_loaders import PyPDFLoader, WebBaseLoader

docs_pdf = PyPDFLoader("ruta/al.pdf").load()      # -> [Document, Document, ...]
docs_web = WebBaseLoader("https://ejemplo.com").load()
```

**Patrones útiles**

- **Iterar páginas** de PDF y preservar contexto: añade claves propias a `metadata` (p. ej., `doc_id`, `section`).
- `lazy_load()` para streams grandes: procesa en pipeline sin cargar todo en memoria.
- **Normalización**: crea una función que homogenice `metadata` (e.g., `source`, `type`, `uri`) para *search logs* y *retrieval filters*.

**Problemas comunes**

- Contenido dinámico no aparece con `WebBaseLoader` → usa un loader con navegador headless.
- Codificaciones raras → asegúrate de `encoding` correcto o limpia con `chardet` / `ftfy`.

---

## 3 · Text Splitters (fragmentar para ventanas de contexto y eficiencia)

**Problema que resuelven:** Los LLMs tienen *context window* limitada (p. ej., ~128k tokens). Documentos largos → errores 413/429, coste alto, olvidos y peor *attention*.

**Qué hacen:** Dividen contenido en *chunks* (fragmentos) *semánticamente razonables*, con **solapamiento** para preservar continuidad.

**Clases comunes y cuándo usarlas**

| Splitter | Cuándo usarlo | Claves de configuración |
|---|---|---|
| `RecursiveCharacterTextSplitter` | Opción por defecto "inteligente" (capítulos → párrafos → oraciones) | `chunk_size`, `chunk_overlap`, `separators` |
| `CharacterTextSplitter` | División rígida por separador simple | `separator`, `chunk_size`, `chunk_overlap` |
| `TokenTextSplitter` | Ajuste por tokens (útil si cobras/limitas por tokens) | `encoding_name`, `chunk_size`, `chunk_overlap` |
| `MarkdownHeaderTextSplitter` | Respeta jerarquía Markdown (#, ##, ###) | `headers_to_split_on` |
| `SentenceTransformersTokenTextSplitter` | Casos con embeddings específicos/tokenización ST | `chunk_size`, `chunk_overlap` |

**Parámetros esenciales**

- `chunk_size`: tamaño del fragmento (caracteres o tokens).
- `chunk_overlap`: solape entre fragmentos (20–15% del `chunk_size` es frecuente).
- `separators`: prioridad de cortes (`\n\n`, `\n`, `.`, `,`, ` `, `""` ...).
- `length_function`: cómo medir longitud (caracteres vs tokenizador).
- `add_start_index`: anota índice inicial en `metadata` para trazabilidad.

**Patrones recomendados**

- **PDF → split por página y luego por texto**: combina `PyPDFLoader` con `RecursiveCharacterTextSplitter`.
- **Tamaños orientativos** (ajusta según caso/RAG store):
  - *Resumenes/LLM direct query*: 1–2k tokens, `overlap` 10–20%.
  - *Embeddings para RAG*: 300–800 tokens, `overlap` 10–15%.
- **Preserva contexto en** `metadata`: `source`, `page`, `chunk`, `start_index`, `section`.
- **Evalúa calidad de chunks**: revisa cortes en tablas/listas; quizá usa splitters específicos (Markdown/HTML).

**Errores y costes**

- Prompt demasiado largo → 413/429 o *rate limit*: reduce `chunk_size`, filtra/recupera top-k antes de invocar el LLM.
- Coste de inferencia alto → *pre-summarization* de chunks + *map-reduce* o *refine*.

**Mini-pipeline canónico**

```python
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

docs = PyPDFLoader("doc.pdf").load()  # 1 Document por página
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=150)
chunks = splitter.split_documents(docs)  # mantiene metadata por chunk
```

---

## 4 · División "inteligente" de información con `RecursiveCharacterTextSplitter`

**Objetivo:** partir documentos largos en *chunks* manejables para LLMs y para indexación (RAG), preservando coherencia.

**Clase principal**

- `from langchain.text_splitter import RecursiveCharacterTextSplitter`

**Parámetros clave**

| Parámetro | Qué controla | Guías prácticas |
|---|---|---|
| `chunk_size` | Longitud aproximada del fragmento (caracteres por defecto) | 800–1200 para embeddings; 2k–4k para prompts directos; ajusta a la *context window*. |
| `chunk_overlap` | Solape entre fragmentos | 10–20% del tamaño del chunk (p. ej., 100–200 sobre 1k). |
| `separators` | Prioridad de corte (capítulo → párrafo → oración → palabra) | Déjalo por defecto o personalízalo si cortas tablas/markdown. |
| `length_function` | Cómo medir longitud | Útil para trabajar en **tokens** (p. ej., tiktoken). |
| `add_start_index` | Añade posición al `metadata` | Facilita trazabilidad y highlights. |

**Patrones de uso**

```python
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=150,
)
chunks = splitter.split_documents(docs)  # preserva metadata por chunk
```

**Por qué el *overlap***
Mitiga cortes en frases/secciones; aporta contexto al siguiente chunk.

**Pitfalls**

- `chunk_size` excesivo → errores 413/429 y coste alto.
- Sin *overlap* → respuestas que "pierden el hilo".
- PDFs: primero por **página** (loader) y luego por **texto** (splitter).

**Patrón "map-reduce" para resúmenes**

1. `map` : resume cada chunk.
2. `reduce` : sintetiza todos los resúmenes en uno final (otra invocación al LLM).
   *Ventaja:* escalable a documentos muy grandes.

---

## 5 · Embeddings: conceptos esenciales

**Definición:** vector numérico de longitud fija que codifica el **significado** de un texto/objeto.
**Idea central:** textos semánticamente similares → vectores cercanos (alta similitud); diferentes → lejanos.

**Propiedades útiles**

- Dimensión fija (p. ej., 384, 768, 1024, 1536, 3072…).
- Distancias comunes: **cosine**, **L2**, **dot-product**.
- Dominio: texto, código, imágenes, audio (según modelo).

**Buenas prácticas**

- **Mismo modelo** para indexar y consultar.

- Versionado del modelo de embeddings: reindexa si cambias de modelo.
- Normalización (unit-length) si la base vectorial/consulta lo requiere.

**Antipatrones**

- Mezclar embeddings de proveedores/modelos distintos en el mismo índice.
- Usar dimensiones muy grandes sin necesidad → coste/latencia y mayor ruido.

---

## 6 · Embeddings con LangChain (OpenAI u otros)

**Interfaz común**

- `from langchain_openai import OpenAIEmbeddings` *(o proveedor equivalente)*
- Todos heredan de `Embeddings` → interoperables con `VectorStore`, `Retriever`, etc.

**Uso mínimo**

```
emb = OpenAIEmbeddings(model="text-embedding-3-large")
v = emb.embed_query("texto de ejemplo")     # → list[float]
V = emb.embed_documents(["t1", "t2", ...]) # batch
```

**Comparación rápida**

| Opción | Ventaja | Considera |
|---|---|---|
| OpenAI (hosted) | Calidad, mantenimiento cero | Coste por uso, dependencia externa |
| Sentence-Transformers (local) | Sin datos a terceros | DevOps, GPU/CPU y memoria |
| Cohere/Azure/Google, etc. | Alternativas con SLAs/regiones | API keys, matices de licencia |

**Cálculo de similitud (ejemplo)**

- Cosine similarity (0–1): cercano a 1 → muy similares.
- Para ranking de resultados recuperados del vectorstore.

**Consejos de producción**

- Cachear embeddings (evitar recomputar).
- Limpiar/normalizar texto antes de embeddar (lower, espacios, símbolos).
- Guardar `metadata` rico (source, page, section, chunk_id, model_version).

---

## 7 · Bases de datos vectoriales (VectorStores)

**Qué son:** almacenes optimizados para vectores + búsqueda por similitud/aproximada (ANN).

**Flujo de INGESTA estándar**

1. **Cargar** documentos (loaders).
2. **Splittear** en chunks (splitter).
3. **Embeddings** de cada chunk.
4. **Indexar** en el vectorstore (con su `metadata`).

**Flujo de CONSULTA estándar**

1. Convertir la **query** a embedding.
2. Buscar **top-k** vecinos (cosine/L2/dot).
3. Recuperar chunks + metadata.
4. (Opcional) **Rerank** con modelo cross-encoder.
5. Pasar contexto "elegido" al LLM.

**Opciones habituales**

| Tipo | Ejemplos | Dónde encaja |
|---|---|---|
| Local/open source | **Chroma**, FAISS, Qdrant, Milvus | Desarrollo, POCs, on-prem, control total |
| Gestionadas (cloud) | **Pinecone**, Weaviate Cloud, Qdrant Cloud | Producción escalable, menor DevOps |

**Parámetros típicos del índice**

| Parámetro | Impacto |
|---|---|
| `metric` (cosine/L2/dot) | Afecta ranking y normalización |
| `dim` | Debe coincidir con el modelo de embeddings |
| Algoritmo ANN (HNSW, IVF, PQ) | Compromiso entre precisión, memoria y velocidad |
| `top_k` | Nº de vecinos; 3–10 suele ser buen punto de partida |

**Buenas prácticas**

- **Metadata filtering**: añade `source`, `doctype`, `page`, `lang`, `tags` para búsquedas filtradas.
- **Mantenimiento**: compactación/reindexado periódico si hay muchas altas/bajas.
- **Privacidad**: si hay datos sensibles, prefiere local/on-prem o cifrado.

**Errores comunes**

- Desalinear `dim` con el modelo → excepción al indexar/consultar.
- Cambiar el modelo de embeddings sin reindexar → *drift* y malos resultados.
- `top_k` muy alto → latencia y coste; muy bajo → omisión de contexto relevante.

---

## 8 · VectorStores con LangChain (Chroma)

**Objetivo:** almacenar *chunks* embebidos y recuperar por similitud.

**Paquetes e imports**

- Instalación: `pip install chromadb`
- Import típico:
  - Hoy: `from langchain_community.vectorstores import Chroma`
  - **Aviso de migración:** pronto → `pip install langchain-chroma` y `from langchain_chroma import Chroma`
- Loaders/splitter/embeddings frecuentes:
  `PyPDFDirectoryLoader`, `RecursiveCharacterTextSplitter`, `OpenAIEmbeddings`

**Creación & persistencia**

| Paso | API | Nota |
|---|---|---|
| Cargar PDFs de una carpeta | `PyPDFDirectoryLoader(path).load()` | Devuelve `List[Document]` |
| Splitear | `splitter.split_documents(docs)` | Preserva `metadata` |
| Embeddings | `OpenAIEmbeddings(model="text-embedding-3-large")` | Usa **el mismo modelo** para index y query |
| Construir índice | `Chroma.from_documents(docs, embedding_function=emb, persist_directory="chroma_data")` | Crea SQLite + carpeta |
| Reabrir índice | `Chroma(persist_directory="chroma_data", embedding_function=emb)` | No reingesta |

**Búsqueda básica**

```
results = vectorstore.similarity_search(query, k=3)      # → List[Document]
# Alternativas: similarity_search_with_score, max_marginal_relevance_search (MMR)
```

**Parámetros prácticos**

- `persist_directory` : ruta del índice en disco.
- `collection_name` : agrupa dominios distintos.
- `k` : vecinos a devolver (3–8 suele ir bien).
- `mmr` (vía `max_marginal_relevance_search` ): más diversidad, menos duplicados.

**Buenas prácticas**

- Añade `metadata` útil: `source` , `page` , `chunk_id` , `start_index` , `doctype` , `person_names` , etc.
- Tamaños de chunk orientativos para RAG con contratos: **1000–5000** chars, `overlap` **200–1000**.
- Reindexa si cambias **modelo** o **parámetros de split**.

**Pitfalls**

- `k` demasiado alto → latencia/ruido.
- Duplicados semánticos → usa **MMR** o Multi-Query Retriever (abajo).
- No mezcles embeddings de distinta dimensión en la misma colección.

---

## 9· Retrievers en LangChain (interfaz estándar)

**Qué son:** envoltorios de búsqueda que devuelven `List[Document]` con una **interfaz unificada**.

**Creación desde un VectorStore**

```
retriever = vectorstore.as_retriever(
  search_type="similarity",                      # "similarity" | "mmr" | "similarity_score_threshold"
  search_kwargs={"k": 2, "score_threshold": 0.0}   # según el tipo
)
docs = retriever.invoke(query)  # también .batch(), .stream()
```

**Search types y kwargs**

| `search_type` | Cuándo usar | `search_kwargs` típicos |
|---|---|---|
| `"similarity"` | Top-k más cercanos | `k` |
| `"mmr"` | Diversidad / menos duplicados | `k` , `fetch_k` (candidatos), `lambda_mult` (diversidad) |
| `"similarity_score_threshold"` | Filtrar por calidad mínima | `score_threshold` , `k` |

**Ventajas frente a `.similarity_search` directo**

- Intercambiable (otro backend u otro tipo de búsqueda sin tocar el resto del pipeline).
- Compatible con *retrievers* avanzados de LC.

**No confundir con:** *Document Loaders* (ingesta). Retriever = **consulta**, Loader = **carga inicial**.

---

## 10 · Multi-Query Retriever (con LLM)

**Qué hace (pipeline):**

1. Reformula la **query** en variantes (sinónimos, subconsultas).
2. Ejecuta búsquedas por cada variante (usando tu retriever base).
3. **Fusiona y desduplica** resultados con ayuda de un LLM.

**Setup mínimo**

```
from langchain_openai import ChatOpenAI
from langchain.retrievers.multi_query import MultiQueryRetriever

base = vectorstore.as_retriever(search_type="similarity", search_kwargs={"k": 2})
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
mqr = MultiQueryRetriever.from_llm(retriever=base, llm=llm)
docs = mqr.invoke(query)
```

**Cuándo usarlo**

- Consultas ambiguas o vocabulario heterogéneo.
- Colecciones con duplicidad alta.

**Costes y control**

- ↑ Llamadas a LLM → **latencia/coste**.
- Limita variantes (si el impl. lo permite) y mantén `k` razonable.
- Úsalo junto a **MMR** para máxima cobertura sin ruido.

---

## 11 · RAG (Retrieval-Augmented Generation)

**Arquitectura canónica**

1. **Retrieve:** `retriever.invoke(query)` → `docs`.
2. **Construcción de prompt:** plantilla con **contexto** (chunks + metadata) + **pregunta**.
3. **Generate:** LLM responde **usando solo el contexto** (instrucción explícita).
4. **Post-proceso:** citar fuentes, verificar cobertura, formatear respuesta.

**Elementos LangChain típicos**

- `PromptTemplate` / `ChatPromptTemplate`
- `RunnableParallel` / `RunnableSequence`
- `StrOutputParser`
- `retriever` (puede ser Multi-Query, MMR, etc.)

**Buenas prácticas de prompt**

- Instrucción de **grounding**: "Responde solo con la información del CONTEXTO; si no está, di que no sabes."
- **Citas**: incluye `source/page/chunk_id`.
- Limita tokens: recorta a los **n** mejores chunks (3–6), ordenados por score/diversidad.

**Mejoras comunes**

- **MMR** o **reranking** (cross-encoder) tras retrieve.
- **Fusionar resúmenes** de chunk para reducir tokens.
- **Filtros por metadata** (fecha, persona, tipo de contrato).

**Métricas de calidad**

- *Faithfulness / Groundedness* (no alucina).
- *Context Precision/Recall* (recupera lo necesario sin ruido).
- Tiempo de respuesta y coste por consulta.

---

## 12 · Arquitectura base de un sistema RAG

**Componentes mínimos**

| Componente | Rol | Claves de implementación |
|---|---|---|
| **VectorStore** (p. ej., Chroma) | Persistencia de embeddings | `Chroma(persist_directory, embedding_function)` |
| **Embeddings** | Index y query | **Mismo modelo** en index/consulta ( `OpenAIEmbeddings` u otro) |
| **Retrievers** | Recuperación de chunks | `vectorstore.as_retriever(search_type=..., search_kwargs=...)` |
| **LLM (consultas)** | Reformulación/derivación y apoyo al retrieve | Modelo económico/determinista (p. ej. `gpt-4o-mini`, `temperature=0`) |
| **LLM (generación)** | Redacción final | Modelo más capaz; `temperature` baja–media |
| **Prompts** | Instrucciones del retrieve y de la respuesta | Separar en archivo `prompts.py` |
| **Cadenas LCEL** | Orquestación | `RunnablePassthrough`, `PromptTemplate`, ` |

**Config centralizada ( `config.py` )**

- Modelos: `EMBEDDING_MODEL`, `QUERY_MODEL`, `GENERATION_MODEL`.
- Chroma: `CHROMA_DB_PATH`.

- Retriever: `SEARCH_TYPE` ( `"mmr"` , `"similarity"` , ...), `SEARCH_K` , `MMR_LAMBDA` (0–1), `MMR_FETCH_K` .

**Buenas prácticas**

- **No "hardcodear"** rutas ni modelos en código de negocio: usa `config.py` o env vars.
- Dos LLMs ≠ obligatorio, pero **recomendable** (coste vs calidad).

---

## 13 · Prompts del sistema

**Tipos sugeridos**

| Prompt | Uso | Consejos |
|---|---|---|
| **RAG main prompt** | Instruye al LLM de respuestas para usar solo el **contexto**; cita fuentes | Variables: `{context}` , `{question}` ; exige estructura y cautela ("si no está, di que no sabes") |
| **Multi-Query Retriever prompt** | Derivar 2–4 consultas alternativas | Especializa al dominio (sinónimos legales, partes, ubicaciones, condiciones) |
| **Relevancia por fragmento** *(opcional)* | Filtrar chunks irrelevantes antes del prompt final | Útil si hay ruido en el índice |
| **Extracción de entidades** *(opcional)* | Añadir `metadata` rico (personas, direcciones, importes, fechas) | Útil tanto en **ingesta** como en **pre-prompt** |

**Antipatrones**

- Prompt único y genérico para todo el pipeline.
- No fijar el **formato de salida** (dificulta UI/parseo).

---

## 14 · Cadena LCEL: orquestación de retriever → prompt → LLM → parseo

**Esqueleto recomendado**

```
rag_chain = (
  {
    "context": final_retriever | format_docs,  # subcadena
    "question": RunnablePassthrough()          # llega en invoke(...)
  }
  | prompt          # PromptTemplate con {context} y {question}
  | llm_generation  # ChatOpenAI u otro
  | StrOutputParser()
)
```

**Puntos finos**

- `RunnablePassthrough()` : "placeholder" para inyectar la pregunta en `invoke` .
- Subcadenas: permite **preprocesar** el output del retriever antes del prompt.
- Devuelve un único **string** (o estructura Pydantic si lo defines).

**Errores típicos**

- Mezclar `Document` directly en prompt sin formateo → contexto confuso.
- Usar distintos modelos de embeddings entre index y query.

---

## 15 · Recuperación híbrida (Ensemble Retriever)

**Idea:** combinar varios retrievers y **agregar puntuaciones** con pesos.

**Setup típico**

```
base = vectorstore.as_retriever(search_type="mmr",
  search_kwargs={"k": SEARCH_K, "lambda_mult": MMR_LAMBDA, "fetch_k": MMR_FETCH_K}
)
simr = vectorstore.as_retriever(search_type="similarity",
```

```
    search_kwargs={"k": SEARCH_K}
)

mqr  = MultiQueryRetriever.from_llm(retriever=base, llm=llm_queries, prompt=multiquery_prompt)

final_retriever = EnsembleRetriever(
    retrievers=[mqr, simr],
    weights=[0.7, 0.3],
    similarity_threshold=0.75,   # opcional: filtra resultados débiles
)
```

**Cuándo usarlo**

- Colecciones con **duplicados** y **vocabulario variado**.
- Queries ambiguas: **MultiQuery** cubre sinónimos; **MMR** aporta diversidad.

**Parámetros clave**

| Parámetro | Efecto | Rango/tip |
|---|---|---|
| `weights` | Importancia relativa por retriever | Suma no tiene por qué ser 1; empieza con 0.7/0.3 |
| `similarity_threshold` | Corta ruido | 0.7–0.8 suele ser buen inicio |
| `SEARCH_K` | Vecinos finales | 2–6 según longitud de chunks |
| `MMR_LAMBDA` | Relevancia vs diversidad | 0.6–0.8 equilibrado |
| `MMR_FETCH_K` | Candidatos previos a MMR | 20–50 típicamente |

**Pitfalls**

- Pesos mal calibrados → o muy redundante o muy disperso.
- `threshold` alto → sin resultados; bajo → ruido.

# Página de Notas del Tema

Esta página está pensada para que puedas anotar ideas clave, dudas y reflexiones importantes sobre el tema anterior.

# Tema 4 – LangGraph

## Guía de referencia

**Usa esta guía como recordatorio rápido de los conceptos y APIs clave.** Está pensada para cualquier proyecto, no solo para los ejemplos del curso.

---

## 1 · Qué es LangGraph (y cómo encaja con LangChain)

| Pregunta | Resumen útil | Implicación práctica |
|---|---|---|
| **¿Sustituye a LangChain?** | No. **Complementa** a LangChain usando sus interfaces (Runnable, TypeDict, etc.). | Puedes combinar **chains LCEL** dentro de nodos de LangGraph. |
| **¿Qué añade?** | **Grafos con ciclos**, **ramificación dinámica** (branching), **estado global compartido**. | Modela bucles, reintentos, bifurcaciones y memoria sin "salirte" a Python puro. |
| **¿Cuándo usarlo?** | Cuando hay **decisiones dinámicas**, **iteraciones**, **memoria persistente**, **colaboración/ paralelismo** entre partes. | Flujos lineales simples → LCEL; flujos complejos → LangGraph (o mixto). |

---

## 2 · Componentes fundamentales

| Componente | Qué es | ¿Qué hace? | Tips |
|---|---|---|---|
| **Estado (State)** | Estructura (normalmente `TypedDict`) compartida por todo el grafo. | **Lee/escribe** datos accesibles por cualquier nodo. | Devuelve **diccionarios parciales**; LangGraph **fusiona** sobre el estado. |
| **Nodo (Node)** | Unidad de trabajo: **función Python** o **Runnable**. | Consume/actualiza parte del estado. | Mantén nodos **cohesivos** (1 responsabilidad). |
| **Arista (Edge)** | Conexión **dirigida** entre nodos. | Define el **flujo** (orden, bifurcaciones, ciclos). | Usa directivas `START` y `END` para entrada/salida. |

---

## 3 · Patrón de arquitectura (paso a paso)

1. **Define el estado** (p.ej., `class State(TypedDict): ...`).
2. **Crea el grafo de estado**: `graph = StateGraph(State)`.
3. **Declara nodos** (funciones o runnables) que **reciben y/o devuelven** partes del estado.
4. **Añade nodos**: `graph.add_node("nombre", funcion_o_runnable)`.
5. **Conecta**: `graph.add_edge(START, "nodoA"); graph.add_edge("nodoA","nodoB"); graph.add_edge("nodoB", END)`.
6. **Compila**: `app = graph.compile()`.
7. **Invoca** con **estado inicial** (dict): `result = app.invoke(initial_state)`.

---

## 4 · Estado: definición, lectura y escritura

| Aspecto | Cómo hacerlo | Detalle clave |
|---|---|---|
| **Definir** | `class State(TypedDict): campo1: str; n: int; ...` | Tipado guía el **merge** de salidas de nodos. |
| **Inicializar** | `initial = {"campo1": "...", "n": 0, ...}` | Solo necesitas lo **mínimo** para arrancar el flujo. |
| **Leer** | En el nodo: acceder como **dict** ( `state["campo1"]` ). | Piensa el estado como **contexto global**. |
| **Actualizar** | El nodo **retorna dict parcial**: `{"campo1": nuevo_valor}` | No necesitas devolver **todo** el estado. |
| **Persistencia** | El estado viaja por todo el grafo. | Útil para **memoria** entre pasos no contiguos. |

**Idea clave:** Cada nodo **devuelve un dict parcial** y LangGraph **lo fusiona** en el estado según el esquema.

---

## 5 · Nodos: funciones y Runnables

| Tipo | Ventaja | Cuándo |
|---|---|---|
| **Función Python** `(state: State) -> dict` | Simplicidad y control total. | Transformaciones, utilidades, E/S liviana. |
| **Runnable de LangChain** | Composición declarativa (LCEL), reutilización. | LLM calls, chains ya definidas, tool-calls. |

**Buenas prácticas de nodo**

- **Pura** si es posible (misma entrada → misma salida).
- **Pequeña y legible** (una responsabilidad).
- **Validación**: si parseas respuestas de LLM, considera **Pydantic**/parsers estructurados.

---

## 6 · Aristas, orden y control de flujo

| Acción | API/Concepto | Nota |
|---|---|---|
| **Fijar inicio/fin** | `add_edge(START, "A")` , `add_edge("Z", END)` | Requisito para la ejecución. |
| **Secuencial** | `A → B → C` | Flujo típico “pipeline”. |
| **Branching** | Arista condicional basada en **estado** | Permite **rutas alternativas** según condiciones. |
| **Ciclos/iteraciones** | **A → B → A** (con condición de parada) | Implementa **bucles y reintentos** dentro del grafo. |

**Diferencia con LCEL**: No estás obligado a “salida del nodo = entrada del siguiente”. Los nodos consultan el **estado global**.

---

## 7 · Compilación y ejecución

| Paso | Qué hace | Tips |
|---|---|---|
| `compile()` | Congela el grafo y valida la conexiones. | Compila **una vez**, reutiliza la app. |
| `invoke(initial_state)` | Ejecuta el flujo con el estado inicial. | Loguea/inspecciona el **estado de salida**. |
| **Estados parciales** | Puedes añadir campos vacíos y llenarlos en nodos. | Útil cuando inputs llegan **más tarde**. |

---

## 8 · Patrón típico de uso de “LLM sobre notas/datos” (generalización)

| Paso | Patrón | Qué cuidar |
|---|---|---|
| **Extracción** | Nodo LLM que extrae **listas** (participantes, temas, acciones…). | Pide **formato explícito** (CSV, JSON). Considera **parsers**. |
| **Agregación** | Nodo que construye **documentos** (acta, resumen) a partir del estado. | **Lee** múltiples claves del estado, **escribe** una. |
| **Normalización** | Funciones auxiliares (limpieza, split, casting). | Mantén la **lógica fuera** del prompt si es determinista. |
| **Transcripción/ETL externo** | Paso de **ASR** o carga de datos como etapa previa. | Encapsula E/S fuera del grafo y **vuelca** al estado. |

> Este patrón sirve para cualquier entrada textual (reuniones, entrevistas, tickets, logs) y cualquier salida estructurada (entidades, decisiones, artefactos).

---

## 9 · Estado avanzado: Annotated Types y reducers

Cuando **varios nodos escriben la misma clave** del estado, por defecto la **última** escritura **sobrescribe** a las anteriores.
Para **acumular/combinar** resultados, define la clave como **Annotated** con una **función reductora**:

| Elemento | Descripción | Ejemplos |
|---|---|---|
| **Tipo anotado** | `Annotated[List[str], reducer]` | La clave sabe **cómo** combinar nuevos valores. |
| **Reducer (reductora)** | Función que combina **valor actual** + **nuevo** → **combinado**. | `operator.add` (listas → concat; números → suma). |
| **Casos típicos** | Logs acumulados, listas de errores, scoring incremental, merges. | Logs: `List[str]` + `operator.add` → se **concatenan**. |
| **Custom reducers** | Función Python propia si lo predefinido no encaja. | Unir **sets** (evitar duplicados), **merge** de dicts por claves. |

**Checklist para reducers**

- Elige el **tipo base** correcto (list, set, dict, int…).
- Selecciona reducer coherente (**add**, **union**, **merge**…).
- Inicializa el estado con el **neutro** del tipo ([], set(), {}, 0…).
- Devuelve desde cada nodo **solo** la clave afectada (p.ej., `{"logs": ["Paso X ok"]}`).

---

## 10 · Buenas prácticas (que escalan)

- **Diseña primero el estado**: nombres claros, tipos útiles, qué produce/consume cada nodo.
- **Nodos idempotentes** cuando sea posible; facilita reintentos y bucles.
- **Bifurcaciones explícitas**: documenta condiciones y claves del estado usadas.
- **Valida salidas de LLM** con **parsers/JSON** cuando vayan al estado.
- **Observabilidad**: usa una clave `logs` (Annotated + reducer) para trazas.
- **Modularidad**: encapsula **E/S externas** (ASR, DB, APIs) fuera y escribe al estado.
- **Composición**: integra **chains LCEL** dentro de nodos cuando convenga.
- **Reutilización**: compila una vez y **reusa** la app para múltiples invocaciones.

---

## 11 · Control de flujo y decisiones (branching)

| Necesito… | Qué uso en LangGraph | ¿Qué hace? | Notas rápidas |
|---|---|---|---|
| **Bifurcar el flujo** según el estado | **Arista condicional** (router/"routing function") | Evalúa `router(state)` y **dirige** a un nodo u otro. | Define un **router** puro (sin E/S externas). |
| **Elegir nodo de destino** por nombre | **Mapping de rutas** | El `router` devuelve una **clave** (p. ej. `"par"`, `"impar"`) que mapea a nombres de nodos. | Mantén **coherencia** entre claves devueltas y nombres de nodo. |
| **Entrar por el principio con decisión** | `START → router(...)` | Condiciona el **primer salto** del grafo. | Útil para **pre-clasificar** casos. |
| **Finalizar** tras cada rama | `add_edge("<rama>", END)` | Cierra la ejecución en cualquier rama. | Puedes reconverger varias ramas antes de `END`. |

**Patrón (esqueleto):**

1. Define estado mínimo (p. ej. `number: int`, `resultado: str`).
2. Declara nodos "hoja" (p. ej., `caso_par`, `caso_impar`) que **solo escriben** el resultado.
3. Declara `router(state)` que devuelve una **clave** de ruta (`"par"` / `"impar"`).
4. Conecta `START` con **arista condicional** (source, `router`, `{ "par": "caso_par", "impar": "caso_impar" }`).
5. Conecta cada rama a `END` (o a pasos posteriores).

**Buenas prácticas**

- El router **no** debe mutar estado; que sea **determinista** y pequeño.
- Prevé **ruta por defecto** (fallback) para entradas inválidas.
- Mantén los nombres de ramas **autoexplicativos** y documenta la **condición**.

---

## 12 · Aristas condicionales (routing) y decisiones

| Necesito… | API/Concepto | ¿Qué hace? | Tips |
|---|---|---|---|
| **Decidir camino en tiempo de ejecución** | `graph.add_conditional_edges(source_node, router_fn, routes_map)` | Llama a `router_fn(state)` y salta al nodo mapeado. | El `router_fn` **no muta** estado; devuelve una **clave** (nombre de ruta). |
| **Enrutar desde el inicio** | `START → router_fn` (vía `add_conditional_edges`) | Primera bifurcación con el **estado inicial**. | Nombra rutas = **nombres reales de nodos**. |
| **Cerrar cada rama** | `graph.add_edge("<rama>", END)` | Finaliza tras la rama. | O re-converge en un nodo común antes de `END`. |

**Esqueleto de router**

```
def router(state: State) -> str:
    # lógica pura en base a `state`
    return "nodo_destino"  # p. ej. "par" / "impar" / "escalado"
```

**Errores comunes**: rutas que **no** existen; router que **modifica** el estado; strings de ruta ≠ nombres de nodos.

---

## 13· Streaming y observabilidad con LangGraph

| Necesito… | API / Parámetro | ¿Qué obtengo? | Cuándo usarlo |
|---|---|---|---|
| **Ver progreso paso a paso** | `app.stream(initial_state, config=..., stream_mode="updates")` | Emite **actualizaciones parciales** del estado por nodo. | Para mostrar "qué cambió" en cada transición. |
| **Ver estado completo tras cada nodo** | `stream_mode="values"` | **Estado completo** después de cada ejecución de nodo. | Debug detallado o auditoría. |
| **Ver tokens/mensajes LLM** | `stream_mode="messages"` | **Tokens/mensajes** generados por LLMs de los nodos. | Inspeccionar prompts/respuestas. |
| **Debug exhaustivo** | `stream_mode="debug"` | Mezcla de valores, updates y mensajes. | Diagnósticos finos (más ruidoso). |

**Patrón básico (loop de streaming)**

```
for chunk in app.stream(state_inicial, config=config, stream_mode="updates"):
    for node_name, salida in chunk.items():
        # p.ej., acumular trazas si `salida` contiene "historial"
```

**Consejos de observabilidad**

- Mantén una clave `historial` (Annotated + reducer) para **trazas** legibles.
- En UI (Streamlit, FastAPI websockets, etc.), **pinta** en tiempo real lo que llegue del stream.
- Limita `stream_mode` a lo necesario (menos ruido = mejor UX).

**Config & threading (para reanudación)**

- Estructura típica: `config={"configurable": {"thread_id": ticket_id}}` .
- El `thread_id` identifica la **ejecución** y permite **reanudación** contra el mismo checkpoint.

---

## 14· Detención y reanudación (Human-in-the-Loop) desde la UI

| Paso | Qué haces | API/Concepto | Detalles clave |
|---|---|---|---|
| **1. Invocar grafo** | Lanzas `stream()` con `state_inicial` y `config` (incluye `thread_id` ). | `app.stream(...)` | Si el grafo tiene `interrupt_before=["procesar_humano"]` , se **detiene** ahí. |
| **2. Leer estado parcial** | Al detenerse, obtienes **estado** + **config** (con identificadores de checkpoint/hilo). | Devuelto tras `stream()` | Guarda en tu UI (p. ej., `st.session_state["tickets"][ticket_id]` ). |
| **3. Solicitar input humano** | Muestras textarea/campo para `respuesta_humano` . | UI (Streamlit) | También puedes mostrar **contexto** recuperado por RAG para ayudar al agente. |
| **4. Actualizar estado persistido** | Inyectas la respuesta en el estado del hilo detenido. | `app.update_state(config, {"respuesta_humano": texto})` | No re-compilas; **actualizas** el estado guardado en SQLite. |
| **5. Reanudar ejecución** | Continúas el grafo desde el punto de pausa. | `app.stream(..., config=config)` o `app.invoke(..., config=config)` | Usa **el mismo** `config` / `thread_id` que al pausar. |
| **6. Cerrar ciclo** | Lees `respuesta_final` , marcas ticket como resuelto (IA u humano), guardas trazas. | Estado final | Muestra fuentes, confianza, tiempos, quién resolvió. |

**Checklist HITL**

- Define pausa con `interrupt_before=["procesar_humano"]` (o `interrupt_after` según convenga).
- Persiste con `SqliteSaver` (o `MemorySaver` si no necesitas durabilidad).
- Usa `update_state(...)` **antes** de reanudar.
- Mantén **idempotencia**: si re-intentas, los nodos deberían tolerarlo.

# Página de Notas del Tema

Esta página está pensada para que puedas anotar ideas clave, dudas y reflexiones importantes sobre el tema anterior.

# Tema 5 – Memoria y Gestión de Contexto con LangChain y LangGraph

## Guía de referencia

**Usa esta guía como recordatorio rápido de mecanismos y parámetros para gestionar memoria y contexto en apps con LLM.** Es general y aplicable a cualquier proyecto (no depende de los ejemplos del curso).

---

### 1 · Conceptos clave (coste, contexto y sesiones)

| Concepto | ¿Qué es? | Por qué importa |
|---|---|---|
| **Context window** | Límite de tokens que el LLM puede procesar por petición. | Define cuánto historial/carácteres puedes enviar; superar el límite da errores o truncado. |
| **Coste por tokens** | La facturación se basa en tokens de entrada/salida. | Más historial ⇒ más coste y latencia. Optimiza memoria y prompts. |
| **Historial de conversación** | Mensajes previos de usuario y asistente. | Mantiene coherencia (p. ej., recordar el nombre del usuario). |
| **Sesiones / threads** | Identificador para aislar conversaciones concurrentes. | Evita mezclar historiales de distintos usuarios/hilos. |
| **Persistencia** | Dónde se guarda la memoria (RAM o disco/BD). | RAM es volátil; disco permite reanudar conversaciones tras reinicios. |

---

### 2 · Patrón básico: historial manual en el prompt (rudimentario)

**Idea:** Mantener una lista `history: List[BaseMessage]` y pasarla a la plantilla con `MessagePlaceholder("history")` .

| Elemento | Cómo se usa | Ventajas | Limitaciones / riesgos |
|---|---|---|---|
| `MessagePlaceholder("history")` | Inserta una variable (lista de mensajes) en el prompt. | Control total del prompt. | Gestión manual tediosa; crece sin control; sin sesiones por defecto. |
| `HumanMessage` , `AIMessage` | Tipos de mensaje para construir `history` . | Estructura clara. | Hay que actualizar `history` tras cada turno. |
| **Bucle de chat** | `chain.invoke({"input": user_input, "history": history})` | Fácil de entender. | Sin aislamiento por sesión; alto coste si el historial crece. |

**Cuándo evitarlo:** apps multiusuario, historiales largos, necesidad de persistencia o control fino de coste.

---

### 3 · LangChain (simple, sin persistencia): `RunnableWithMessageHistory`

**Para** prototipos o apps sencillas sin necesidad de guardar en disco.
**Clase:** `langchain_core.runnables.history.RunnableWithMessageHistory`

#### Piezas esenciales

- **Store en RAM:** diccionario `store: Dict[str, InMemoryChatMessageHistory]` .
- **Historial por sesión:** `InMemoryChatMessageHistory` (no persistente).
- **Factory de historial:**

```
def get_session_history(session_id: str):
    if session_id not in store:
        store[session_id] = InMemoryChatMessageHistory()
    return store[session_id]
```

- Envoltura con memoria:

```
chain_with_memory = RunnableWithMessageHistory(
    chain,                                # tu Runnable base (prompt + LLM)
    get_session_history,                  # cómo obtener el historial
    input_messages_key="input",           # campo del input del usuario
    history_messages_key="history"        # placeholder en tu prompt
)
```

- Config por sesión (aislamiento):

```
config = {"configurable": {"session_id": "<id-sesión>"}}
resp = chain_with_memory.invoke({"input": user_input}, config=config)
```

### Pros / Contras

| Pros | Contras |
|---|---|
| Sencillo; añade memoria y sesiones con poco código. | RAM volátil; sin persistencia; menos flexible para flujos complejos. |
| Compatible con prompts existentes. | No apto para cargas concurrentes/robustas a producción. |

**Checklist rápido**

- Define `get_session_history`
- Envuelve tu cadena con `RunnableWithMessageHistory`
- Pasa `session_id` vía `config["configurable"]`

---

## 4 · LangGraph (recomendado): `MemorySaver` + `MessageState`

**Para** control del flujo y memoria integrada en grafos, con checkpoints.

### Conceptos LangGraph

| Elemento | Descripción | Uso típico |
|---|---|---|
| `StateGraph` | Grafo de nodos con estado compartido. | Orquestar pasos (nodos) del agente. |
| `MessageState` | Estado predefinido con `messages: List[BaseMessage]` . | Mantener historial sin crear un estado custom. |
| **Checkpointer** | Guarda el estado tras cada nodo. | Inspección, reanudación, memoria. |
| `MemorySaver` | Checkpointer en **RAM** (no persistente). | Memoria rápida entre invocaciones del grafo. |

### Patrón mínimo

```
# 1) Definir grafo con estado de mensajes
workflow = StateGraph(StateSchema=MessageState)

# 2) Nodo de chat (lee historial y añade respuesta)
def chatbot_node(state: MessageState):
    sys = SystemMessage(content="Eres un asistente útil…")
    msgs = [sys] + state["messages"]
    resp = llm.invoke(msgs)
    return {"messages": [resp]}  # se concatena al estado

workflow.add_node("chatbot", chatbot_node)
workflow.add_edge("START", "chatbot")

# 3) Compilar con checkpointer en RAM
memory = MemorySaver()
app = workflow.compile(checkpointer=memory)

# 4) Invocar por sesión (thread_id)
config = {"configurable": {"thread_id": "<id-sesión>"}}
```

```
result = app.invoke({"messages": [HumanMessage(content=user_input)]}, config=config)
```

### Ventajas clave

- **Sesiones robustas** mediante `thread_id`.
- **Checkpointing** automático (mejor inspección y reanudación).
- **Performance**: menos sobrecarga de "pegado" manual del historial.

### Considera persistencia

- `MemorySaver` es volátil. Para persistir, cambia el checkpointer (p. ej., SQLite/Redis) en lugar de `MemorySaver`.

---

## 5 · Ventana deslizante (control de coste): `trim_messages`

**Objetivo:** Enviar al LLM solo los **N últimos turnos** (manteniendo el historial completo en estado/checkpoint si quieres), para controlar tokens y latencia.

**Función:** `langchain_core.messages.trim_messages`

| Parámetro | Qué hace | Notas |
|---|---|---|
| `strategy="last"` | Mantiene los últimos N "tokens" según contador. | Útil para ventana fija. |
| `max_tokens=N` | Límite usado por el contador. | Con contador "por mensaje", N ≈ nº de mensajes. |
| `token_counter=len` | Forma de contar tokens. | Con `len` sobre cada mensaje ⇒ cada mensaje cuenta como 1. |
| `start_on="human"` | Asegura que el primer mensaje recortado sea humano. | Recomendable en chats. |
| `include_system=True` | Preserva el `SystemMessage`. | Mantiene el rol/instrucciones. |

**Patrón de uso dentro del nodo**

```
# Preparar el trimmer (una vez)
trimmer = trim_messages(
    strategy="last",
    max_tokens=4,          # p. ej., últimos 4 mensajes
    token_counter=len,     # cuenta cada mensaje como 1
    start_on="human",
    include_system=True,
)

def chatbot_node(state: MessageState):
    sys = SystemMessage(content="Eres un asistente…")
    # Recortar SOLO para invocar el LLM
    trimmed = trimmer.invoke(state["messages"])
    msgs = [sys] + trimmed
    resp = llm.invoke(msgs)
    return {"messages": [resp]}  # El estado aún guarda TODO el historial
```

**Buenas prácticas**

- Mantén **estado completo** para auditoría, pero **recorta** para invocar el LLM.
- Ajusta `max_tokens` (o cambia a contador de tokens real) según el modelo/plan.
- Usa junto a checkpointers persistentes si necesitas reanudar sesiones largas.

---

## 6 · ¿Qué enfoque usar?

| Caso | Enfoque recomendado | Motivo |
|---|---|---|
| Prototipo rápido, 1 usuario, sin reanudar | `RunnableWithMessageHistory` + `InMemoryChatMessageHistory` | Simplicidad. |
| Chat multiusuario con control de flujo | **LangGraph** ( `StateGraph` + `MessageState` ) | Orquestación + memoria integrada. |
| Memoria no persistente, rendimiento | LangGraph + `MemorySaver` | Rápido y aislado por `thread_id` . |
| Conversaciones largas / control de coste | Añade `trim_messages` (ventana deslizante) | Limita tokens enviados. |
| Reanudación tras reinicio/escala | Checkpointer **persistente** (SQLite/Redis/…) | Estado durable. |

---

## 7 · Memoria persistente con LangGraph (SQLiteSaver)

**Objetivo:** conservar historiales entre ejecuciones, reinicios o despliegues.

| Pieza | ¿Qué es? | Opciones/Detalles |
|---|---|---|
| **Checkpointer** | Mecanismo de guardado tras cada nodo del grafo. | Se pasa al compilar: `app = graph.compile(checkpointer=...)` . |
| `MemorySaver` | Checkpointer **en RAM** (volátil). | Útil para prototipos, no persiste. |
| `SQLiteSaver` | Checkpointer **persistente** en disco. | Recomendado para conservar chats/hilos. |
| **Conexión** | `sqlite3.connect("historial.db", check_same_thread=False)` | `check_same_thread=False` permite acceso concurrente controlado. |
| **Sesiones** | `config={"configurable":{"thread_id": "<id>"}}` | Aísla historiales por chat/hilo. Reusar `thread_id` reanuda la conversación. |

**Patrón mínimo:**

1. **Estado y grafo**

```
workflow = StateGraph(StateSchema=MessageState)
def chatbot_node(state):
    sys = SystemMessage("…instrucciones…")
    msgs = [sys] + state["messages"]
    resp = llm.invoke(msgs)
    return {"messages": [resp]}
workflow.add_node("chatbot", chatbot_node)
workflow.add_edge("START", "chatbot")
```

1. **Persistencia**

```
conn = sqlite3.connect("historial.db", check_same_thread=False)
cp = SQLiteSaver(conn)
app = workflow.compile(checkpointer=cp)
```

1. **Invocación por sesión**

```
config = {"configurable": {"thread_id": chat_id}}
app.invoke({"messages": [HumanMessage(content=user_input)]}, config=config)
```

**Buenas prácticas**

- Un `thread_id` por chat; un `user_id` (si lo modelas) para agrupar chats.
- Rotación/backup de la DB (vacuum, índices básicos).
- Si necesitas HA/escala: cambia a Postgres u otro checkpointer persistente equivalente.

---

## 8 · Memoria vectorial — conceptos y diseño

**Idea:** almacenar información en una **BD vectorial** y recuperar **fragmentos relevantes por similitud semántica**, en lugar de enviar todo el historial.

| Decisión | Opciones | Comentarios |
|---|---|---|
| **Ámbito** | Por-chat, por-usuario (global), por-equipo | Lo normal: **global por usuario** + por-chat opcional. |
| **Embeddings** | OpenAI, Cohere, local (e5, Instructor…) | Elige por calidad/coste; homogeneidad entre index y query. |
| **Store** | Chroma, FAISS, PGVector, Milvus, Weaviate | Chroma: simple y persistente en disco local. |
| **Schema** | `document` , `id` , `metadata` (p. ej., `user_id` , `source` , `type` ) | Etiqueta: `type` ∈ {profile, preference, work, location, …}. |
| **Retriever** | `k` (topK), distancia (cosine, l2), filtros por metadatos | Empieza con `k∈[3,8]` ; filtra por `user_id` . |
| **Fusión con memoria secuencial** | Ventana deslizante + vectorial | Vectorial aporta **recuerdo duradero**; ventana controla coste del turno. |

**Cuándo usarla**

- Chats largos/antiguos donde no compensa arrastrar todo el hilo.
- Perfilado progresivo de usuario (gustos, nombre, ubicaciones, rol…).
- Búsquedas "no literales" (sin depender de coincidencia exacta).

---

## 9 · Caso práctico (abstracto): combinar memoria volátil por chat + memoria vectorial global

**Arquitectura lógica**

- **LangGraph +** `MemorySaver` → historial **activo** del chat (barato y rápido).
- **Vector store (Chroma)** → **memoria global** del usuario (perfil, gustos, datos duraderos).
- **Nodo de chat**:
    1. Lee último `HumanMessage` (la consulta).
    2. **Recupera** memorias globales relevantes ( `k` fijo, filtra por `user_id` ).
    3. **Inyecta** memorias a `SystemMessage` (o como contexto adicional).
    4. Invoca LLM con `[System] + (ventana deslizante del chat) + [Último Human]` .
    5. **Actualiza**: añade `AIMessage` al estado + (opcional) **extrae** del último mensaje humano datos para **guardar** en vectorial.

**Funciones típicas**

- `save_memory(text, user_id, type=…)` → añade `document` , `id=uuid4()` , `metadata` .
- `search_memory(query, user_id, k=5)` → `vector_store.query` + filtros.
- **Heurística de guardado** (rápida) o **Clasificador con LLM** (mejor calidad) para decidir qué persistir:
    - Nombre: "me llamo…"
    - Ubicación: "vivo en… / soy de…"
    - Profesión: "trabajo como… / soy …"
    - Preferencias: "me gusta… / me encanta…"

**Consejos de implementación**

- Mantén **comandos de depuración** (p. ej., `memorias` ) para listar lo guardado.
- Sanitiza/normaliza texto (minúsculas, trimming, deduplicación por hash).
- Añade **TTL** o políticas (importancia, frescura) si temes crecimiento descontrolado.
- Registra **fuente** (chat_id, timestamp) en metadatos.

# Página de Notas del Tema

Esta página está pensada para que puedas anotar ideas clave, dudas y reflexiones importantes sobre el tema anterior.

# Tema 6 – Agentes de IA y herramientas externas con LangChain y LangGraph

## Guía de referencia

**Usa esta guía como recordatorio rápido de conceptos, opciones y APIs clave.** Está pensada para cualquier proyecto, no solo para el ejemplo del curso.

---

### 1 · Concepto de "herramienta" (tool) en LLM apps

**Idea central:** una *tool* es funcionalidad externa invocable por el LLM para hacer algo más allá de generar texto (p.ej., consultar una BD, ejecutar Python, buscar en la web).

| Concepto | Qué es | Por qué importa |
|---|---|---|
| **Tool** | Función externa (con nombre, descripción y esquema de E/S) que el **LLM puede invocar**. | Permite que el modelo *decida* cuándo usar capacidades externas. |
| **Tool vs. nodo de grafo** | En un grafo (LangGraph) tú ordenas el flujo; con *tools* el **LLM decide** invocar. | Mayor flexibilidad y autonomía del agente. |
| **Tool calling nativo** | Modelos modernos generan **llamadas estructuradas** (JSON) a tools (no texto libre). | Integración más estándar y segura. |
| **Argument schemas** | Tipos y validaciones (normalmente vía Pydantic/type hints). | El LLM sabe **qué parámetros** enviar y de **qué tipo**. |
| **Tool artifacts** | Datos extra que **no** se devuelven al LLM pero quedan disponibles para lógica posterior. | Útiles para registrar metadatos, resultados intermedios o salidas pesadas (p.ej., coordenadas detecciones). |

---

### 2 · Tool calling moderno vs. enfoque *legacy*

| Enfoque | Cómo funcionaba | Problemas | Alternativa actual |
|---|---|---|---|
| ***Legacy* (texto libre)** | Instruías al LLM para que devolviera un *"necesito usar X con argumentos Y"* en texto. | Frágil, parseo propenso a errores. | |
| **Actual (tool calling)** | El LLM devuelve una **estructura JSON**: `tool_name`, `arguments`, `call_id`. | — | Definir tools con **esquemas** y dejar que el LLM las invoque. |

**Claves de definición para tool calling efectivo**

- **Nombre corto y preciso.**
- **Descripción orientada a uso** ("qué hace" y "cuándo usarla").
- **Parámetros tipados** (type hints / Pydantic): obligatorios, opcionales, formatos.
- **Salida bien definida** (string/objeto) y, si aplica, **artifacts** separados.

---

### 3 · Herramientas integradas y registro (LangChain)

**Reutiliza tools existentes cuando sea posible.** Ej.: **Python REPL** (intérprete Python) de `langchain-experimental.utilities.PythonREPL`.

| Paso | Qué haces | Detalle clave |
|---|---|---|
| **Instalar** | `pip install langchain-experimental` | Módulo experimental con utilidades. |
| **Crear instancia** | `python_repl = PythonREPL()` | Inicializa el intérprete. |
| **Registrar como tool** | `Tool(name=..., description=..., func=python_repl.run)` | `name` y `description` guían al LLM para elegir esta tool. |
| **Invocar manualmente** | `tool.run("print(2+2)")` | Útil para testear la tool sin LLM. |

**Campos importantes al registrar ( `Tool` )**

- `name` : único y descriptivo (p.ej., `python_repl` ).
- `description` : explícita, con *cuándo usarla* (p.ej., "ejecuta código Python para cálculos/lógica").
- `func` : *callable* que ejecuta la acción (p.ej., `python_repl.run` ).

**Consideraciones**

- El **LLM elegirá** esta tool si su descripción *encaja* con la tarea.
- **Seguridad**: REPL puede ejecutar código arbitrario → sandboxing/guardrails en producción.

---

## 4 · Crear herramientas personalizadas con `@tool` (recomendado)

**Ruta preferida** por simplicidad y legibilidad.

| Elemento | Requisito | Motivo |
|---|---|---|
| **Decorador** | `from langchain_core.tools import tool` | Convierte una función en tool. |
| **Type hints** | **Obligatorios** en parámetros y retorno | El LLM infiere tipos y valida entrada/salida. |
| **Docstring** | Describe **qué hace** y **cuándo usarla** | Se usa como `description` de la tool. |

**Esqueleto recomendado**

- Firma: `@tool(name="...")` *(opcionalmente sin nombre para usar el de la función)*
- Parámetros tipados (p.ej., `query: str` )
- `-> str | dict | ...` retorno tipado
- Docstring clara: *"Consulta la base de usuarios..."*

**Opciones útiles del decorador**

| Opción | Qué hace | Cuándo usarla |
|---|---|---|
| `name="..."` | Cambia el nombre por defecto (el de la función). | Nombres más amigables o compatibles con tu convención. |
| `return_direct=True` | El **resultado final** de la ejecución será la salida de la tool (no vuelve al LLM). | Flujos donde no quieres post-procesar en el LLM (p.ej., descarga de archivo). |
| `response_format=...` | Permite devolver **salida para el LLM** + **artifacts** separados. | Cuando necesitas registrar/encadenar metadatos pesados. |
| `args_schema=MyModel` | Define manualmente el esquema de entrada (Pydantic). | Validaciones avanzadas, alias, formatos específicos. |
| `infer_schema=True` | Deja que infiera esquema desde type hints. | Rápido prototipado si los hints son suficientes. |

**Inspección rápida (debug)**

- `tool_instance.name` → nombre efectivo
- `tool_instance.description` → descripción efectiva

---

## 5 · Alternativa: `StructuredTool`

**Más flexible en algunos casos**, pero más verboso. Útil si quieres **construir la tool a partir de una función** sin decorarla.

| Cómo | Ejemplo mental | Notas |
|---|---|---|
| `StructuredTool.from_function(fn=...)` o `StructuredTool(**kwargs)` | Envolvente explícito sobre una función ya existente. | Mantén type hints y docstring en la función base para que el esquema sea correcto. |
| Interfaz estándar | `.run(**kwargs)` | Igual que con `@tool`. |

**Cuándo preferir** `StructuredTool`

- Necesitas **fabricar tools dinámicamente** (p.ej., a partir de un catálogo de funciones).
- Quieres **separar** definición de función y construcción de la tool por motivos de arquitectura.

---

## 6 · Patrón de diseño y buenas prácticas

- **Diseña pocas tools, bien descritas.** Evita solapamientos; las descripciones deben **diferenciar claramente** las capacidades.
- **Valida inputs** (type hints/Pydantic) y maneja errores con mensajes **útiles para el LLM**.
- **Idempotencia** y **efectos colaterales controlados**: si la tool escribe/borra, pide confirmaciones o usa *dry-run*.
- **Observabilidad**: registra `tool_name`, `arguments`, `duration`, `artifacts` y resultado.
- **Seguridad**:
  - Limita lo que el REPL puede ejecutar (listas blancas, timeouts, recursos).
  - Sanitiza entradas que viajan a SQL/APIs.
  - Mantén **quotas/reintentos** y *circuit breakers* para APIs externas.
- **UX del agente**: descripciones con *triggers* claros ("si necesitas cálculo aritmético…", "si debes consultar usuarios por ID…").
- **Testing**: prueba tools **aisladas** ( `tool.run(...)` ) y en **bucle LLM** (simulando invocaciones reales).

---

## 7 · Usar herramientas con LLMs en LangChain (tool calling básico)

**Objetivo:** habilitar que el modelo **decida** si llama a una tool y con qué argumentos.

| Paso | Qué haces | API/Detalle |
|---|---|---|
| 1) Crear el LLM | Instancia un chat model determinista. | `ChatOpenAI(model="gpt-4o-mini", temperature=0.2)` (o equivalente del proveedor que **soporte tool calling**). |
| 2) Tener tools | Pueden ser personalizadas ( `@tool` ) o integradas. | Asegura **type hints** + docstring; nombre y descripción claros. |
| 3) Asociar tools al LLM | El LLM "sabe" qué herramientas puede usar. | `llm_with_tools = llm.bind_tools([mi_tool, ...])` |
| 4) Invocar al LLM | Si el modelo necesita una tool, devolverá `tool_calls` (JSON estructurado). | `resp = llm_with_tools.invoke("…")` → `resp.tool_calls` (lista con `name`, `args`, `id`). |
| 5) Ejecutar la tool | Ejecuta tú la tool con esos `args`. | Preferible `tool.invoke(args_dict)` (acepta el JSON) — `tool.run(texto)` solo para una cadena simple. |
| 6) Devolver al LLM (opcional) | Para cerrar la tarea con contexto. | Re-invoca el LLM **incluyendo historial** + resultado de la tool como mensaje/observación. |

**Reglas y trucos**

- **Nombres de tools** (especialmente con OpenAI): alfanumérico y **guiones bajos**, **sin espacios** ni símbolos raros. Si incumples, el proveedor **lanza error**.
- **Descripción** debe decir **qué hace** y **cuándo usarla**; el LLM la usa para decidir.
- `tool.run` **vs** `tool.invoke`:
  - `run(input_str)` → rápido para tests, **solo un string**; no gestiona `args` complejos.
  - `invoke({"arg1":…, …})` → acepta el **payload estructurado** que devuelve el LLM.
- **LCEL (cadenas)**: puedes encadenar `llm_with_tools` y luego extraer `tool_calls` (p. ej., con `operator.attrgetter("tool_calls")` o utilidades de LCEL) y mapear a `tool.invoke` para un flujo más compacto.

## 8 · Devolver un “artefacto” para uso interno (respuesta + artifact)

**Objetivo:** que la tool devuelva **dos salidas**: una para el modelo y otra para tu app (logs, metadatos, binarios ligeros, etc.).

| Qué configurar | Cómo funciona | Nota |
|---|---|---|
| `response_format` en la tool | Indica que habrá **contenido para el LLM** + **artifact** para uso interno. | Implementa el `return` como **tupla** `(contenido, artifact)`. |
| Firma tipada | Usa `Tuple[str, int]`, `Tuple[str, dict]`, etc. | Importa `Tuple` de `typing`. |
| Ejecución | `tool.run(...)` **solo** devuelve el **contenido**. | Para inspeccionar artifact, usa `tool.invoke({...})` o procesa el mensaje estructurado. |

**Ejemplo mental**

- **Contenido:** “Resultado de la consulta SQL …” (entra al LLM).
- **Artifact:** `{"elapsed_ms": 123, "rows": 42}` (no entra al LLM; lo usas en observabilidad/post-procesos).

---

## 9 · Herramientas predefinidas e integraciones (LangChain)

**LangChain** incluye un gran catálogo de tools (búsqueda, ofimática, nubes, DBs, mensajería, etc.).

| Categoría | Ejemplos | Cómo se usa |
|---|---|---|
| **Búsqueda/Web** | Bing, Google, Tavily, Wikipedia | Suele requerir **API key** y cliente. Instancia el wrapper y **ejecuta** con `.run` o **enchaína** con tu LLM. |
| **Docs/Ofimática** | Google Sheets/Docs, Microsoft 365 | Autenticación OAuth o API key; define scopes y crea la tool (normalmente en `langchain_community.tools/...`). |
| **Datos/DBs** | SQL, Mongo, Elasticsearch | Configura conexión; tools para **consultar**/resumir tablas. |
| **Mensajería/Comms** | Slack, Twilio, Gmail (borradores) | Requiere credenciales; útiles para agentes que **actúan** (enviar aviso, crear borrador). |
| **Cálculo/Code** | Python REPL, interpretes | **Peligroso** en producción: sandbox, timeouts y whitelists. |

**Patrón típico de uso**

1. Instalar paquete(s) externo(s) necesario(s) (`pip install …`).
2. Importar wrapper y/o tool pre-hecha.
3. Configurar con claves/parámetros.
4. Probar con `.run(...)`.
5. Añadir a `llm.bind_tools([...])` para que el agente/LLM pueda decidir usarla.

---

## 10 · Fundamentos de los agentes de IA (LangChain / LangGraph)

**Definición:** sistema dirigido por un LLM que **elige acciones** (tools) y su **orden** para lograr un objetivo.

| Componente | Rol | Claves |
|---|---|---|
| **LLM** | “Cerebro” que decide el siguiente paso. | Debe soportar **tool calling**. |
| **Toolkit** | Conjunto de tools disponibles. | Limita y **describe bien** cada tool. |
| **Plantilla de razonamiento** | Patrón **ReAct** (Pensar→Actuar→Observar→… →Responder). | Viene predefinida en agentes de LangChain/LangGraph. |
| **Memoria** | Historial/estado para iterar correctamente. | Conversacional o específica de tarea. |
| **Ejecutor del agente** | Orquesta el bucle pensamiento/acción/observación. | En LangChain y LangGraph hay **agentes predefinidos** para no codificar el loop a mano. |

**Ciclo de un agente**

1. **Instrucción** del usuario →
2. **Pensamiento** (LLM con prompt de agente) →
3. **Elección de tool** + **args** →
4. **Ejecución** de la tool →
5. **Observación** (resultado vuelve al contexto) →
6. Repite 2–5 hasta **respuesta final**.

**Buenas prácticas**

- Toolkit **pequeño y bien diferenciado**.
- Mensajes de error **útiles** para que el agente corrija.
- **Guardrails**: límites de pasos, coste, timeouts, rate limiting.
- **Trazabilidad**: log de `tool_calls`, duración, artifacts y resultados.

---

## 11 · Primer agente con LangChain (tool-calling + ejecución)

**Objetivo:** crear un agente que razone, elija tools y ejecute pasos automáticamente.

| Elemento | API/Opción | ¿Qué hace? | Notas clave |
|---|---|---|---|
| **LLM** | `init_chat_model("gpt-4o-mini", temperature=0)` o `ChatOpenAI(...)` | Modelo con **tool calling** nativo. | Usa temperatura baja para comportamiento estable. |
| **Toolkit** | p. ej. `GmailToolkit().get_tools()` | Colección de tools listas (buscar, leer, crear borradores…). | Evita especificar tool por tool; el agente elige. |
| **Tools extra** | `@tool(...)` funciones propias | Añade capacidades específicas del negocio. | Combina toolkit + tools personalizadas en una lista. |
| **Prompt de agente** | `ChatPromptTemplate` con **system** + `{input}` + `{agent_scratchpad}` | Define pasos, criterios y formato. | `agent_scratchpad` es **obligatorio** en agentes LC: almacena pensamiento/acciones/observaciones intermedias. |
| **Agente** | `create_tool_calling_agent(llm, tools, prompt)` | Prepara un agente ReAct que puede invocar tools. | Requiere LLM con tool-calling. |
| **Ejecución** | `AgentExecutor(agent=..., tools=..., verbose=True, handle_parsing_errors=True, max_iterations=K)` | Orquesta el bucle pensar→actuar→observar. | Define **límite de iteraciones** para evitar bucles/costos. |
| **Invocación** | `executor.invoke({"input": "…"})` | Corre el flujo completo y devuelve `output`. | Pasa **historial** si necesitas contexto conversacional. |

**Buenas prácticas**

- **Nombre y descripción** de cada tool: deben dejar claro **cuándo** usarlas.
- **Errores**: `handle_parsing_errors=True` reintenta automáticamente pasos frágiles.
- **Coste/seguridad**: limita herramientas peligrosas (REPL), añade timeouts/quotas.

---

## 12 · Añadir tools personalizadas al agente (patrones útiles)

**Caso:** extender un toolkit con una tool propia (p. ej., responder en el mismo hilo de un sistema de mensajería/correo).

| Paso | API/Opción | Detalle |
|---|---|---|
| Definir tool | `from langchain_core.tools import tool` → `@tool(name="create_reply_draft")` | Usa **type hints** y un **docstring** que documente: qué hace, *cuándo* usarla y cada argumento. |
| Acceso a APIs | Usa wrapper/cliente del servicio (p. ej., `toolkit.api_resource`) | Mantén la lógica de autenticación/config en un sitio. |
| Cuerpo del mensaje | Construye payload (headers, codificación, referencias) | Ej.: añadir `thread_id` / `in_reply_to` cuando el servicio lo requiera para **responder en hilo**. |
| Añadir al agente | `tools = toolkit.get_tools() + [create_reply_draft]` | El agente la podrá elegir según la descripción. |

**Checklist de una tool sólida**

- **Args** bien tipados (obligatorios vs opcionales).
- **Errores** descriptivos (ayudan al agente a corregir).
- **Idempotencia** o `dry_run` si hay efectos (envíos/altas).
- **Artefactos** si necesitas telemetría (duración, IDs creados).

---

## 13 · Agentes con LangGraph (recomendado para producción)

**Objetivo:** mismo patrón ReAct pero con estado y orquestación más estables.

| Elemento | API/Opción | ¿Qué hace? | Notas |
|---|---|---|---|
| **Agente ReAct** | `from langgraph.prebuilt import create_react_agent` | Crea un agente listo para tools. | Sustituye a `create_tool_calling_agent` en LC. |
| **Memoria de checkpoints** | `from langgraph.checkpoint.memory import MemorySaver` | Guarda/recupera estado entre pasos/llamadas. | Cambiable por almacenes persistentes. |
| **Prompt** | parámetro `prompt=` (system prompt) | Define el comportamiento. | **No** necesitas `{agent_scratchpad}` manual. |
| **Config** | `config={"configurable": {"thread_id": "…"}}` | Aísla sesiones y reanuda estados. | Útil para multiusuario o sesiones largas. |
| **Estado** | Mensajes en `state["messages"]` | Entrada: `HumanMessage` (o `{"role":"user","content":"…"}`) | La salida se lee del **último mensaje** del estado. |

**Diferencias vs LangChain**

- Menos pegamento manual (scratchpad/ejecutor) y **mejor control de estado**.
- API cambia con más frecuencia: revisa parámetros actualizados (p. ej., `prompt=` en lugar de `state_modifier=`).

---

## 14 · Sistemas multiagente con LangGraph Supervisor

**Patrón:** un **supervisor** selecciona/agrega respuestas de **agentes especializados**.

| Componente | API/Opción | Función |
|---|---|---|
| Agentes especialistas | `create_react_agent(model, tools=[...], prompt="Eres especialista en X", name="…")` | Uno por área (búsqueda, cálculos, extracción, redacción…). |
| Supervisor | `from langgraph_supervisor import create_supervisor` | Recibe la petición, **decide** a quién delegar y cuándo parar. |
| Modelos | Puedes usar **modelos distintos** por agente/supervisor | Optimiza coste/rendimiento por tarea. |
| Compilación | `graph = create_supervisor(...); compiled = graph.compile()` | Genera el grafo ejecutable. |
| Ejecución | `compiled.invoke({"messages":[... ]})` | El estado final incluye trazas y mensajes. |

**Topologías**

- **Supervisor** (recomendada): 1 coordinador + N expertos.
- **Red**: agentes colaboran entre sí.
- **Jerárquica**: varios niveles (más compleja y frágil).

**Guardrails**

- Límite de **pasos/iteraciones**, presupuesto, y tiempos.
- Trazabilidad: log de `tool_calls`, latencias, errores, artifacts.

---

## 15 · Arquitectura de una solución multiagente "product-ready"

**Capas clave (independientes del dominio):**

1. **Interfaz & API**

- **API REST** para ingestión de eventos/tareas (POST JSON), **healthcheck**, y consulta de estados.
- Cliente UI (dashboard) que lea estados/alertas, trace pasos y resultados.

1. **Orquestación**

- **Supervisor + especialistas** (LangGraph).
- **Colas**/jobs opcionales si ingestión es alta (no bloquees al cliente).
- **Retentativas** y **backoff** para tools externas.

1. **Integraciones**

- **Búsqueda web** (proveedor search), **inteligencia** (p. ej., análisis/escáner), **mensajería/correo** (notificaciones).
- Normaliza **errores** y **rate limits** por proveedor.

1. **Observabilidad**

- Logs estructurados (agente, tool, args resumidos, duración, coste).
- Trazas por **thread_id** y correlación con IDs de negocio.
- Métricas: latencia por tool, % reintentos, % éxito.

1. **Seguridad y cumplimiento**

- **Sandboxes** para ejecución de código.
- **Escapes**/sanitización (SQL, URLs, prompt injection).
- **Controles de salida** (validación de decisiones "riesgosas").

1. **Persistencia**

- Estados de LangGraph (checkpoints), resultados, y artefactos.
- Configuración por **entorno** (dev/staging/prod).

# Página de Notas del Tema

Esta página está pensada para que puedas anotar ideas clave, dudas y reflexiones importantes sobre el tema anterior.