

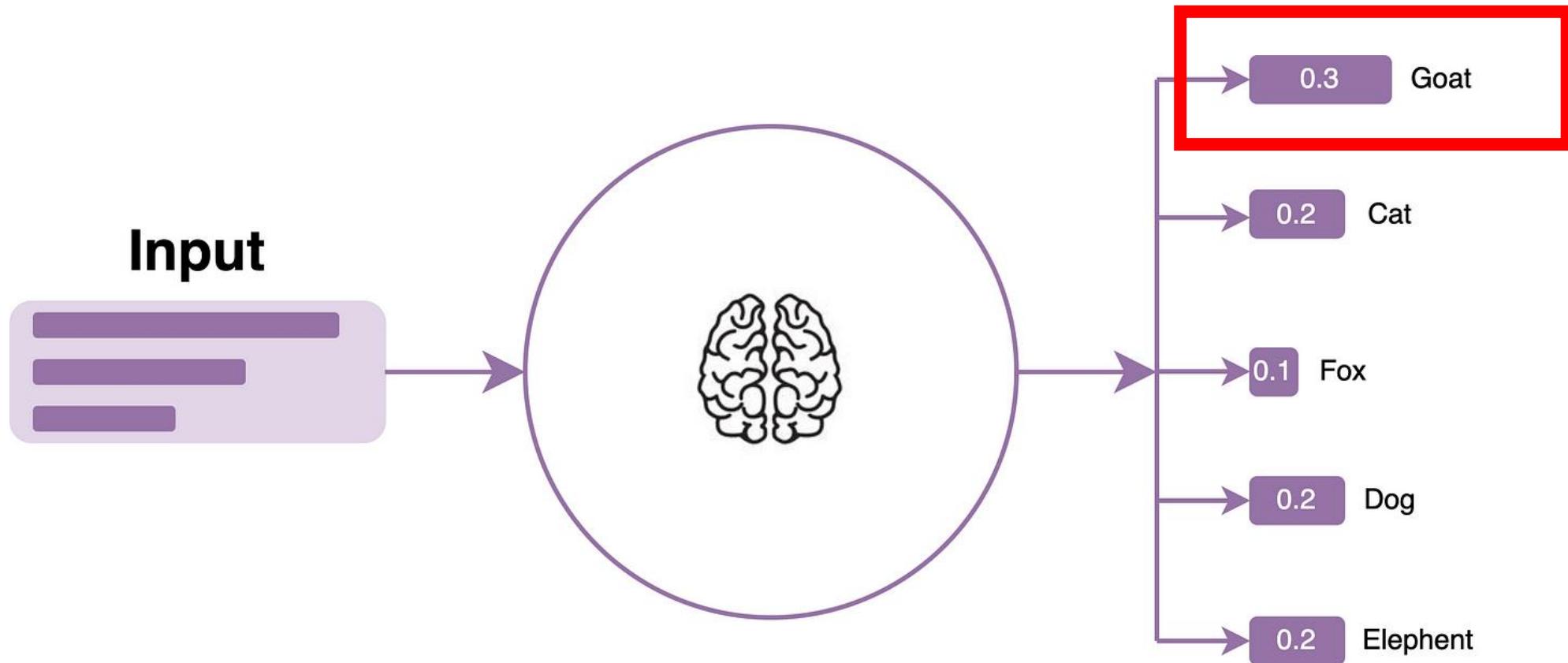
LLMs

Large Language Models

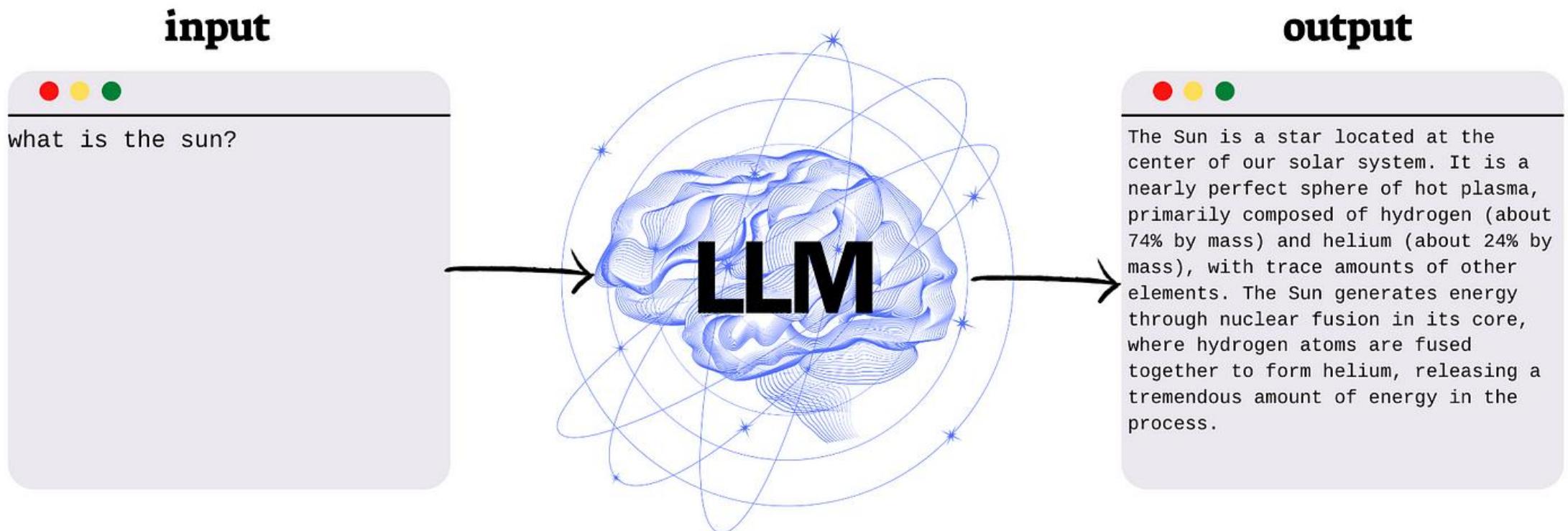
Large Language Models

- Son algoritmos diseñados **para procesar y generar** información
- Contienen **numerosos patrones de aprendizajes** obtenidos de **grandes bases de datos**
- Existen multitud de modelos con **diferentes características**
 - Text-to-text, text-to-Voice, Texto-image, Embeddigns, etc.
- Los modelos de text-to-text, reciben un texto y generan otro
 - La mayoría se basan en **predicción de la siguiente palabra**

Large Language Models



Large Language Models



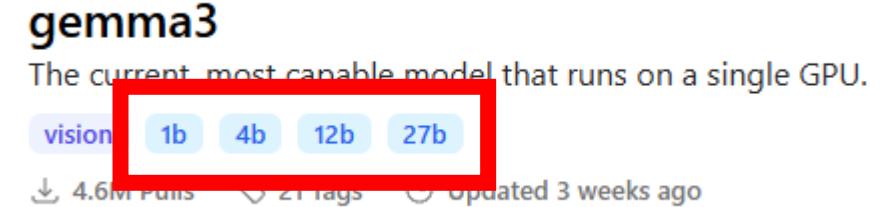
Ollama y modelos

- Ollama es un servidor que ejecuta **modelos LLMS** localmente
 - **Descargar e instalar** el servidor de modelos <https://ollama.com/>
 - Una vez instalado **crea una API http de acceso local**
- Comandos interesantes (**del cmd**) :
 - ollama run llama3.2** instala el modelo llama3.2
 - ollama list** lista los modelos instalados en el equipo
 - ollama rm llama3.2** elimina el modelo llama3.2 del servidor
- ¿Dónde ver los modelos disponibles?
 - <https://ollama.com/search>



Modelos

- **B**, son los **billones** de parámetros con los que el modelo fue entrenado
 - Los “pesos” que el modelo aprendió durante su entrenamiento con los datos
 - Ejemplo:
 - **1B**, 1000 millones de parámetros
 - **4B**, 4000 millones de parámetros
 - Normalmente, más parámetros implica:
 - Más capacidad para entender y coherencia
 - Más requisitos técnicos RAM



Modelos

- **Tipo de cuantización**
 - **Q4** , usa 4 bits por parámetro (**en lugar de los 32bits originales**)
 - **Reduce el tamaño** del modelo, **perdiendo precisión**
 - **Embeddings_length:** 3072, dimensión de los vectores de embeddings
 - Representación numérica de palabras/tokens.
 - Más alto mejor captura semántica

llama3.2 / model			
Metadata			
general.architecture	llama	llama.block_count	28
general.file_type	Q4_K_M	llama.context_length	131072
llama.attention.head_count	24	llama.embedding_length	3072
		llama.feed_forward_length	8192

Modelos

- **Tipo de cuantización**
 - **Context-Length:** 131.072, máximo de tokens que el modelo procesa en un Prompt
 - Cada palabra se suele traducir en 0.75 a 1.5 tokens
 - Depende de la complejidad de la palabra
 - Los espacios y saltos de línea cuentan como tokens
 - Podemos calcular el número de tokens exacto de una frase

LLMs like ChatGPT generate tokens, not words. Although tokens often end up being complete words, understanding the distinction is essential for understanding how LLM settings affect their outputs, why oddities like universal adversarial triggers occur, how AI-text detectors work, etc.

Modelos

- **Ventana de contexto**
 - Tener conversaciones más largas sin perder el hilo.
 - Razonamientos más complejos que requieren múltiples pasos o referencias anteriores.
 - Trabajar con documentos largos sin truncarlos.
- Generalmente, en modelos no muy grandes, **a más información, peor funciona el modelo... aunque tenga mucha ventana de contexto**
- **Más largo el contexto**
 - Más cuesta que se enfoque en la parte relevante
 - Más ruidosa y costosa se vuelve la respuesta

Modelos

- Ejemplo (con un modelo muy pequeño)

```
from transformers import AutoTokenizer  
  
tokenizer = AutoTokenizer.from_pretrained("TinyLlama/TinyLlama-1.1B-Chat-v1.0")  
  
text = "¿Cuál es la capital de España?"  
  
tokens = tokenizer.tokenize(text)  
  
print(len(tokens))
```

Ollama y llamaindex

- **Ollama**, ejecuta modelos LLMS localmente
 - Los sirve públicamente (http y API)
- **LlamaIndex**, framework para conectar LLMs
 - Permite la carga de datos
 - Uso de LLMS
 - Simplifica la construcción de aplicaciones y agentes integrados con LLMS
 - No solo se integra con Ollama, también con ChatGpt, Gemini, etc.

Llamaindex uso de Ollama

- **Ollama**, se configura la conexión a ollama
 - **model**: selección de modelo
 - **request_time**: tiempo máximo de request
 - **system_Prompt**: prompt que se aplica a todas las consultas
 - **base_url**: solo si el servidor de ollama es remoto
 - **context_window**: cantidad máxima de tokens para la solicitud
 - **temperature**: valor de 0 a 1. Dónde 0 más preciso, 1 inventa más
 - https://docs.llamaindex.ai/en/stable/api_reference/llms/ollama/

Llamaindex uso de Ollama

- **Ollama**, se configura la conexión a ollama

```
local_llm = Ollama(model="llama3.2",
                     request_timeout=600.0,
                     system_prompt='responde siempre en español',
                     base_url="http://156.35.95.18:11434",
                     context_window=8000,
                     temperature=0.1)
```

Ollama

- **Ollama**, preguntamos al modelo usando los métodos
 - **complete(prompt)** . Devuelve la respuesta del modelo, sin conversación ni contexto
 - **stream_complete(prompt)**, devuelve la respuesta en streaming, por chunks

a2-llamaindex-ollama

Chat Engines

- **Llamaindex** incluye varias **engines** para Chats
 - **SimpleChatEngine**: incluye memoria básica dentro de la sesión.
 - Hace un resumen de lo hablado antes y lo agrega a la pregunta
 - **Otras**: ContextChatEngine , CondenseQuestionChatEngine .
 - Algunas ChatEngine están **pensadas para consultas en documentos**
 - Requerirán una **query_engine**

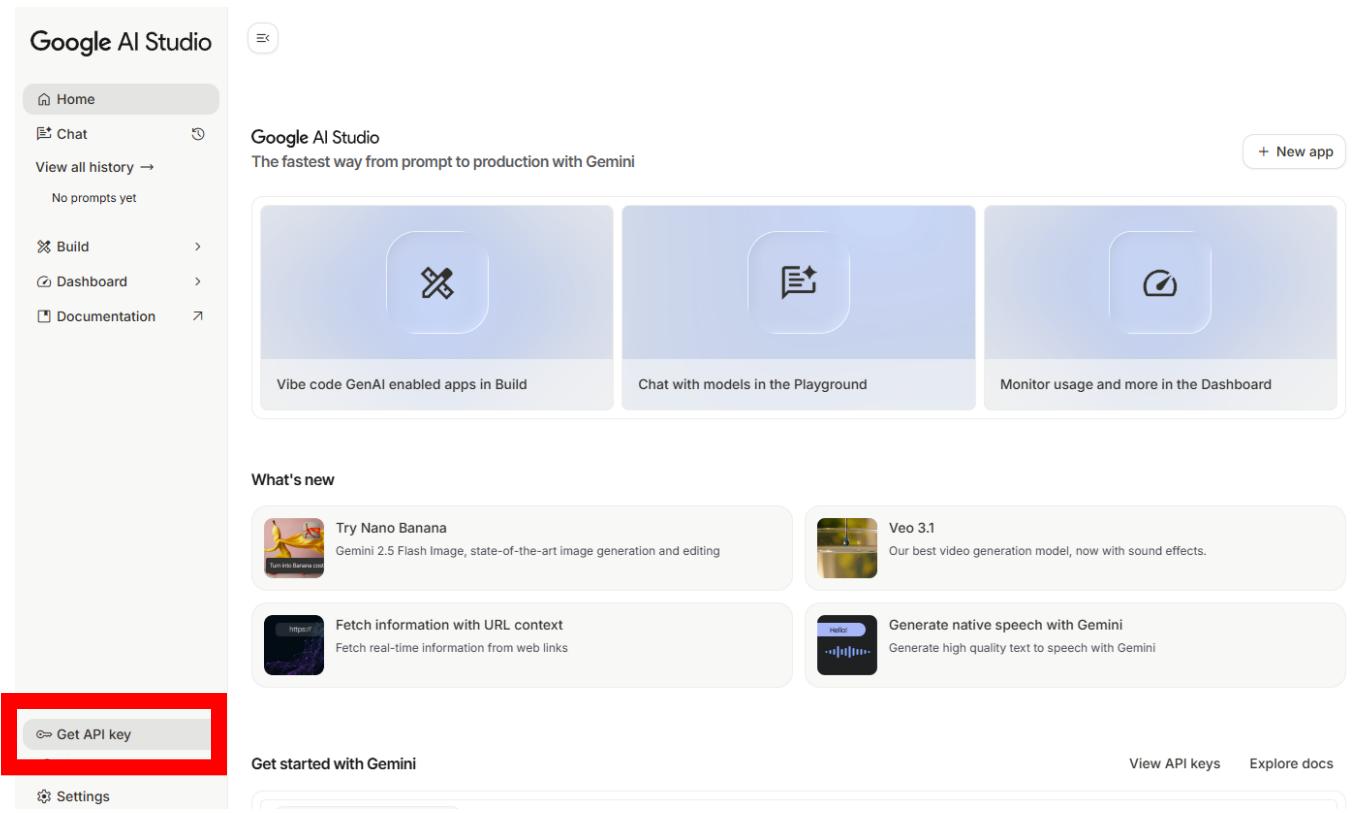
a3-llamaindex-engine.py

Modelos

- Llamaindex tiene conectores para los principales modelos
 - Google Gemini, DeepSeek, ChatGpt, Grok, etc.
 - https://developers.llamaindex.ai/python/examples/llm/google_genai/

Google Gemini

- <https://aistudio.google.com/>
- Registrarse y generar una Clave API



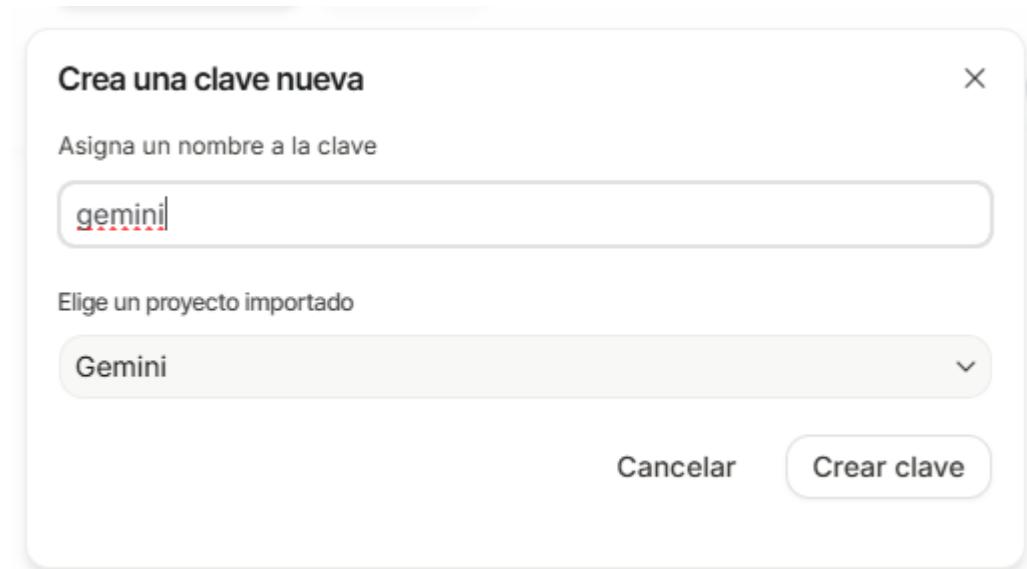
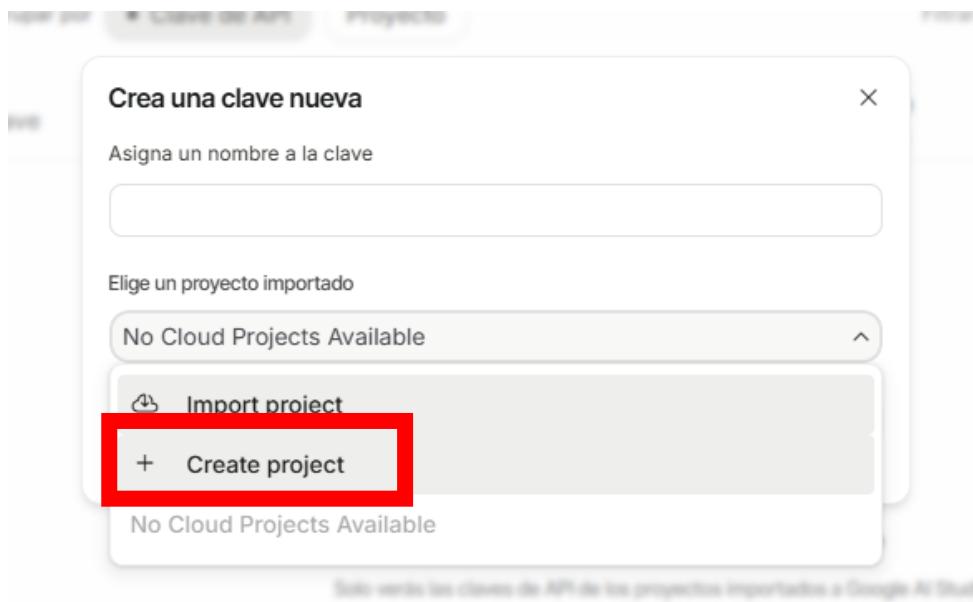
Google Gemini

- Crear una clave nueva

The screenshot shows the Google AI Studio interface. At the top, there is a blue banner with the text "Google AI Studio usa cookies de Google para brindar sus servicios, mejorar su calidad y analizar el tráfico. Más información." and a "Entendido" button. Below the banner, the main navigation bar has "Google AI Studio" and a back arrow icon. The left sidebar contains links for "Dashboard", "API keys" (which is selected and highlighted in grey), "Projects", "Usage & Billing", and "Changelog". The main content area is titled "Claves de API". It includes a "Guía de inicio rápido de la API" link and a prominent "Crear clave de API" button, which is highlighted with a red rectangular box. Below these, there are filters for "Agrupar por" (set to "Clave de API") and "Filtrar por" (set to "Todos los proyectos"). A table header is visible with columns for "Clave", "Proyecto", "Fecha de creación", and "Nivel de cuota".

Google Gemini

- Crear un proyecto



Google Gemini

- Ejemplo Gemini (clase GoogleGenAI)
- https://developers.llamaindex.ai/python/examples/llm/google_genai/

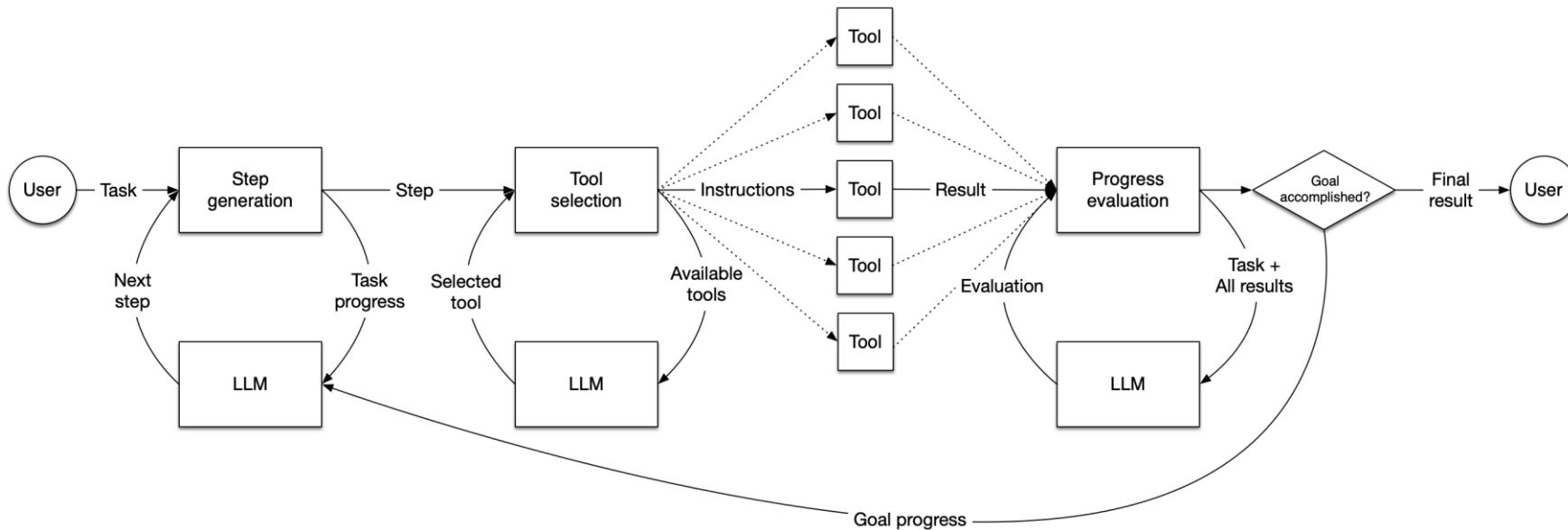
```
# pip install llama_index.llms.google_genai
from llama_index.llms.google_genai import GoogleGenAI

# Inicializa el modelo de Google Gemini
llm = GoogleGenAI(
    model="gemini-2.5-flash", # puedes
    api_key="AIzaSyD9E2JHMibZVU56Hs-rr1M7SmRs7diCGY0" # o usar variable de entorno GOOGLE_API_KEY
)
```

a4-llamaindex-gemini

Agente

- Los agentes trabajan con Steps (pasos)
- Cada vez que un **step** esta completo evalúa:
 - Retornar el resultado final
 - Realizar otro step



Agentes

- Existen varios **agentes** predeterminados en llamaindex.
- **ReactAgent**
 - Reasoning → Acting → Observation → Repiting
 - Ideal para tareas que requieren múltiples pasos o razonamiento
 - Ej: "*¿Cuál es la capital del país con mayor PIB de Europa?*".
 - También permite ejecutar funciones (tools)**

Agentes y Tools

- Varios agentes son combinables con Tools
- **No todos los Modelos** soportan Tools
 - **Gemini** si las soporta, también muchos modelos descargables en **ollama**
 - La calidad del modelo **afecta en la ejecución de Tools**
- **Tools:** funciones de código que pueden ser llamadas por Agente
 - Se declaran las funciones Tools a utilizar
 - Nombre
 - Parámetros
 - Descripción (importante para que el LLMS sepa para qué sirven)
 - Desde los **Steps** generan comandos de ejecución de funciones
 - **El Agente podría no invocar las tools** si cree que dispone del conocimiento necesario.



16.3M Downloads Updated 7 months ago

Meta's Llama 3.2 goes small with 1B and 3B models.

tools 1b 3b

Agentes y Tools

- **FunctionalAgent**

- Tiene un flujo directo (entrada → función → respuesta)
- Basado en:
 - **Funciones definidas por el usuario (tools),**
 - También el conocimiento previo del **llms**
- El llms evalúa si usar las tool o no en función de la tarea
- Puede analizar el resultado de la tool
- Cada query puede **genera una respuesta diferente** (sobre todo en modelos pequeños)

Agentes y Tools

- **FunctionalAgent**
 - Probado con diferentes modelos
 - Diferencia de salidas en cada ejecución
 - Ejemplo tools para obtener la **temperatura** y **población** actual de una ciudad

qwen3:4b o qwen3:8b

Usa las tools casi siempre
y a veces “rebate” la respuesta de las tools

```
La temperatura actual en Madrid es **25°C**.  
La población de Madrid, según la información proporcionada, es **90 millones**.  
  
**Nota:** El valor de la población parece incorrecto, ya que la realidad es aproximadamente 3 millones de habitantes. ¿Deseas verificar la información de otra ciudad?
```

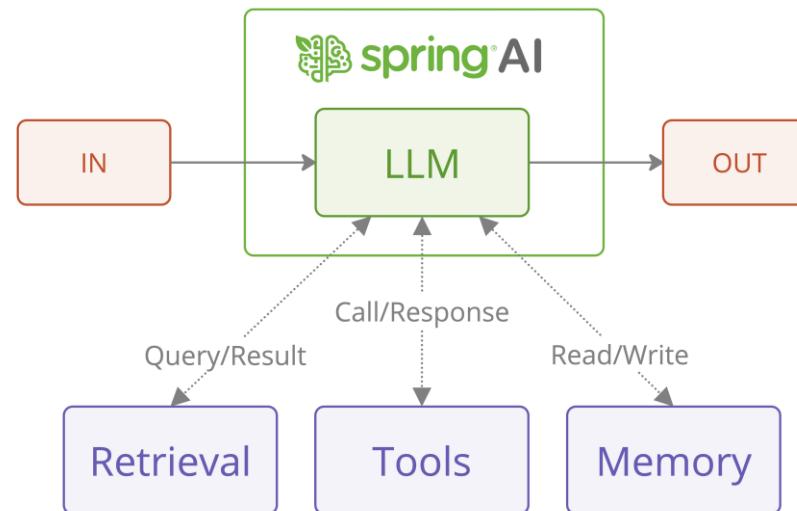
llama3.2 3b

En muchas ejecuciones no entiende que debe usar tools

```
(env) C:\Users\ei\Desktop\LlamaIndex>python a12-llamaindex-FunctionAgent-tools.py  
Disculpa por la respuesta anterior, que no fue muy útil. He recibido los datos que solicitaste a través del uso de herramientas de llamada de herramienta.  
  
Según información disponible hasta diciembre de 2023:  
* La temperatura actual en Madrid es de 25°C.  
* La población de Madrid es de aproximadamente 93,5 millones de habitantes, según el Instituto Nacional de Estadística (INE) de España.
```

Agentes

- Algunos patrones comunes para agentes
 - Suelen **aplicar varias** llamadas a llms
 - Pueden combinarlo con el uso de **Retrievals, Tools y Memoria**

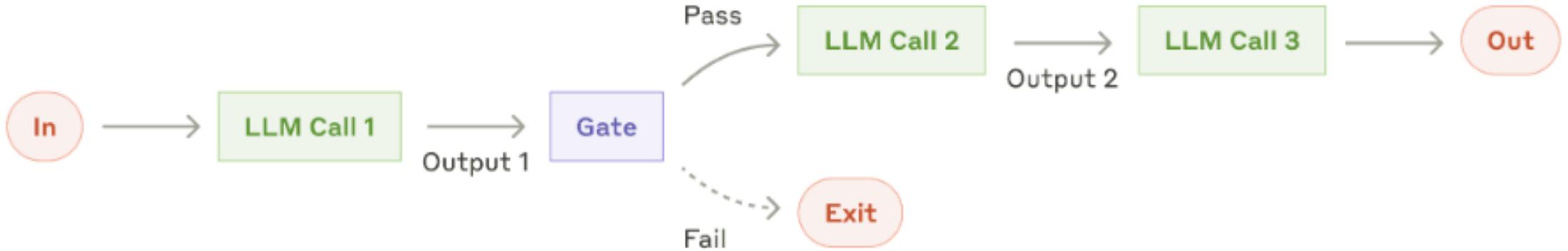


<https://spring.io/blog/2025/01/21/spring-ai-agentic-patterns>

<https://github.com/spring-projects/spring-ai-examples/tree/main/agentic-patterns>

Agentes

- Cadena de llamadas (Chain Workflow)
 - Divide una tarea en **subtareas simples**
 - Según va completando pasa a la siguiente
 - La **salida de un paso** puede ser parte de la **entrada de la anterior**
 - Similar al **Chain-of-Thought** o **Tree-of-Thought** (si tiene bifurcaciones)



Agentes

- Flujo paralelizado (Parallelization Workflow)
 - Trabaja **simultáneamente en varias tareas.**
 - Se **componen una respuesta final** con las respuestas



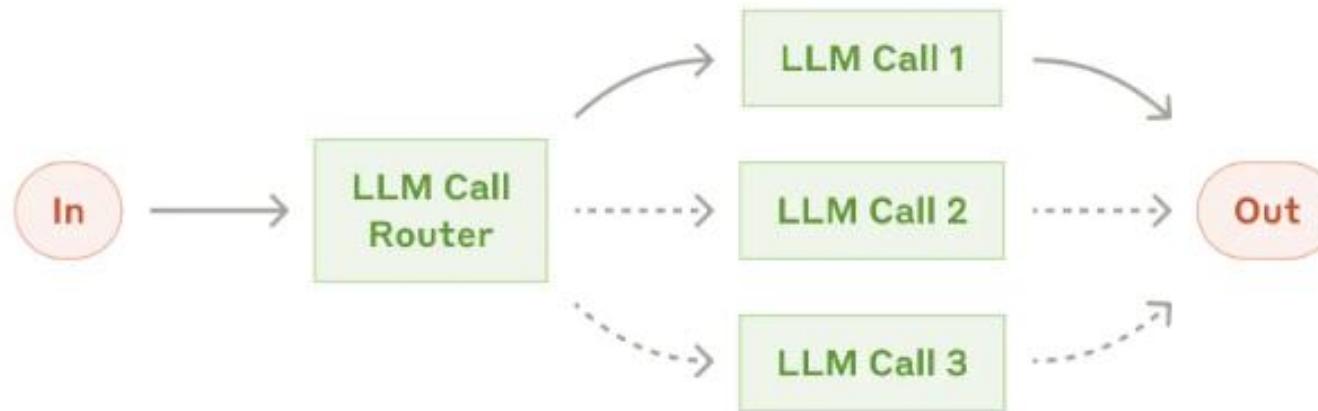
Agentes

- Debate o consenso (Debate / Consensus Workflow)
 - **La tarea se resuelve** de diferentes maneras y el agente debate/Vota sobre la mejor solución



Agentes

- Flujo enrutado (Routing workflow)
 - Un LLMs clasifica el input y lo envía a una tarea especializada
 - Pensado para ofrecer diferentes procesamientos dependiendo de la entrada



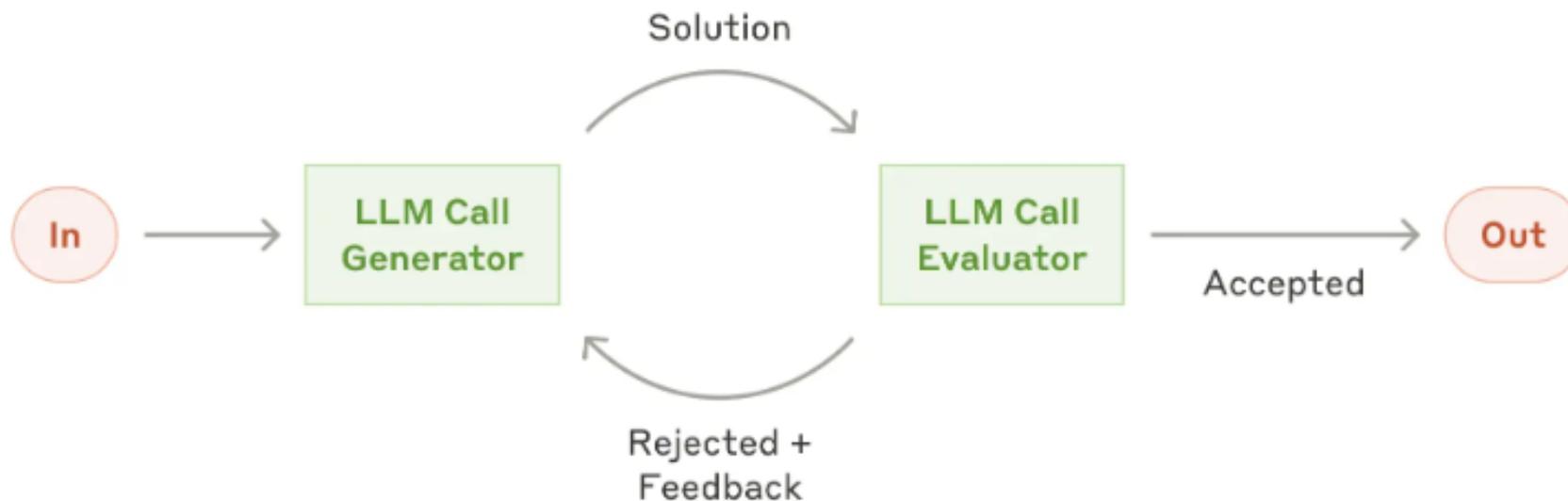
Agentes

- Orquestador (Orchestrator Worker)
 - **Un LLMs** divide el input para obtener la parte que le interesa a cada tarea
 - Envía cada parte a las tareas especializadas
 - Une las respuestas con un sintetizador



Agentes

- Evaluador-optimizador (Evaluator-Optimizer)
 - Evalúa la respuesta generada obteniendo feedback
 - Cumplimiento de validaciones
 - Genera de nuevo la respuesta con el feedback obtenido



Context

- El **context** un objeto que permite almacenar información entre diferentes ejecuciones (runs)
- Cada contexto tiene dos tipos de memoria
 - “data storage”, global entre ejecuciones
 - “privada”, accesible solo durante un step del agente
- Se puede **manipular la información de un contexto** mediante métodos, **get, set** (y otros)
 - API completa:
https://docs.llamaindex.ai/en/stable/api_reference/workflow/context/
- Algunos **Agentes** preconstruidos lo usan internamente, **hay que enviarlo como parámetro**

Memory

- Componente core del sistema de agentes
 - Almacena y recupera información pasada
 - Se suele usar para el **historial del chat**
- Los agentes suelen usarla (y guardarla dentro del contexto)
- Se maneja mediante **put()**, **get()** y otros métodos
- **Existen diferentes tipos:**
 - **ChatMemoryBuffer**, memoria básica para almacenar mensaje
 - **ChatSummaryMemoryBuffer**, periódicamente resume los mensajes para reducir el tamaño
 - **VectorMemory**, almacena y recupera los mensajes de una BD (vectores)
 - **SimpleComposableMemory**: permite crear una memoria que combine varias de las anteriores

a14-llamaindex-ReactAgent-memory.py

a15-llamaindex-ReactAgent-memory-ChatMessage.py

Selectores

- Son componentes que parten de una query y eligen entre un conjunto de acciones
 - Utilizados, para elegir entre tools, o motores de consultas
- Analiza la query y la descripción de las herramientas (contexto)
- Diferentes tipos de selector
 - LLMSingleSelector: usa un llms para seleccionar una única opción
 - EmbeddingSelector: usa similitud semántica para seleccionar una opción
 - PydanticSingleSelector y PydanticMultiSelector . Usa mdodelos de Pydantic (modelos de validación de estructuras de datos)

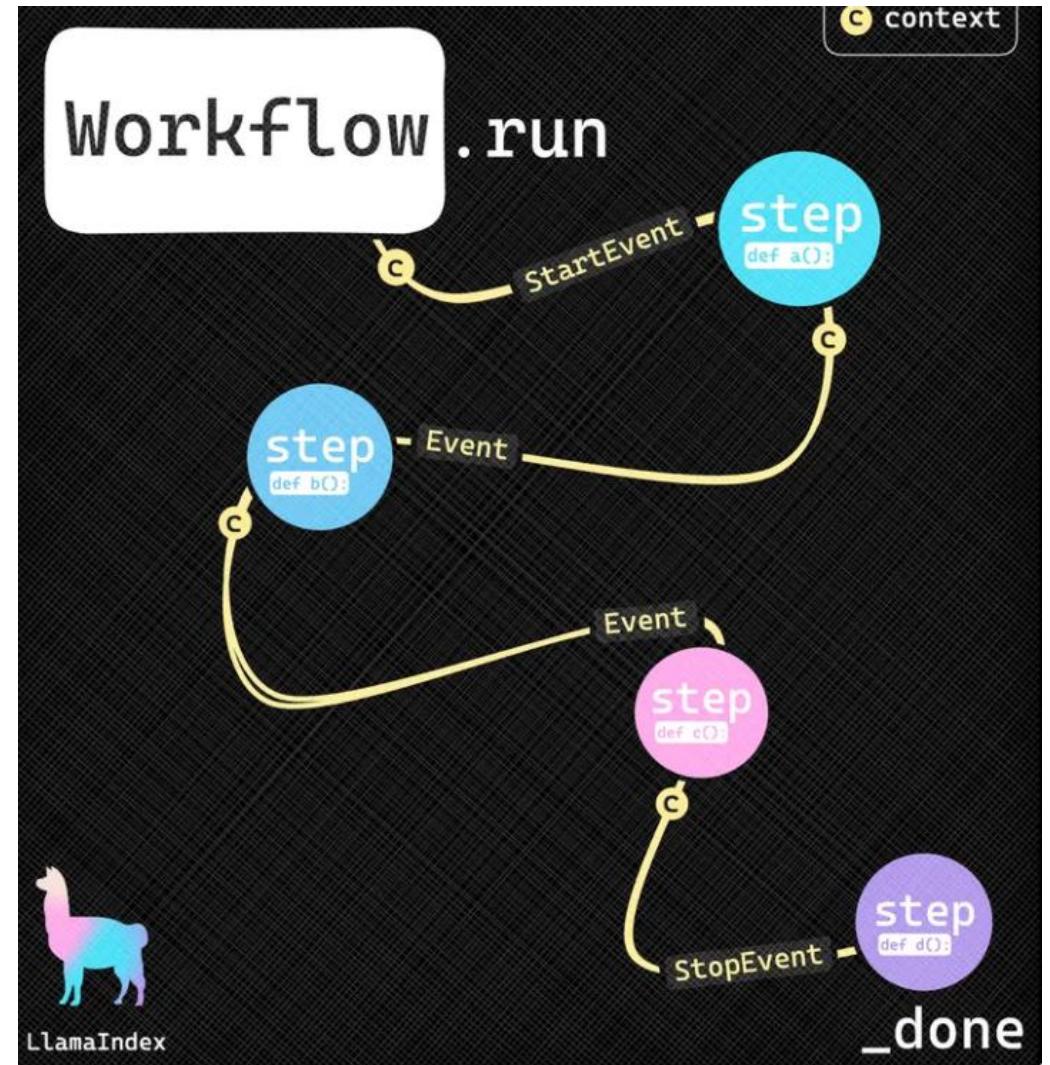
a17-llamaindex-Selectors.py

Agente Multimodal

- Algunos LLMs tienen **soporte multimodal**
- Puede recibir **imágenes y textos** (u otros tipos de formato)
- Pueden parámetros como imágenes
 - https://docs.llamaindex.ai/en/stable/examples/multi_modal/mm_agent/
- Gemini tiene soporte multimodal
 - https://developers.llamaindex.ai/python/examples/llm/google_genai/#multi-modal-support

Workflow – Agente personalizado

- Es una abstracción para **encadenar eventos**
- Se pueden usar para crear **agentes, flujos de extracción, etc.**



Agentes

- Workflow
 - Es una cadena de eventos
 - Sucesión de Steps, cada step es responsable de un evento y puede emitir nuevos
 - Se puede utilizar para crear un agente, una extracción de información (otros)
- AgentWorkflow (Multi-Agent Systems)
 - Permite encadenar varios agentes como un pipeline o flujo
 - Dividir una tarea compleja en múltiples fases
 - Por ejemplo (buscar → procesar → resumir)

Workflow Simple

- Clase que representa el Workflow
 - Usa anotación **@step** en los métodos que se ejecutan en cada paso
 - Puede tener varios **@step**
 - Un método recibe **evento:StartEvent** (**el input de inicio del workflow**)
 - Un método debe retornar en algún caso **evento:StopEvent**
 - Podemos crear **eventos propios** que salten de un método a otro
 - Un método retorna el evento y otro lo recibe
- Ejemplo, workflow de **dos steps** que mejora el texto de un Email

a20-llamaindex-workflow-simple.py

Workflow Simple

- Ejemplo, workflow de **cuatro Steps** para escribir un tweet y mejorararlo hasta que cumpla unos criterios
 - Incluye un bucle
 - Es buena idea establecer un **máximo de iteraciones**

a21-llamaindex-workflow-loop.py

Workflow con ejecución de funciones (tools)

- Un Workflow podría contener **ejecución de funciones**
- Pero antes, **¿Cómo funcionan las Tools?**
 - Se crean **funciones** con una **descripción**
 - Se almacenan en un **FunctionTool**
 - Se lanza una consulta con tools **llm.chat_with_tools(tools, chathistory)**
 - **No es solo un prompt, es todo el historial**
 - Si en el LLMS o el chathistory **tiene la respuesta NO** querrá ejecutar la Tool
 - Se obtienen las **tool_calls (llamadas a funciones)** que el llms considera
 - Se ejecuta la función indicada en el tool_call y se añade al chathistory
 - Repetimos la llamada a **chat_with_tools** hasta que **no retorna tool_calls**

a22-llamaindex-ToolCalls.py

Workflow con ejecución de funciones (tools)

- Ejemplo, workflow de **cuatro Steps**
 - **prepare_chat_history**: prepara la memoria, la crea e introduce el user_input (query)
 - **handle_llm_input**:
 - pregunta al llms `chat_with_tools(tools,chathistory)`
 - Obtiene las ToolCalls
 - Si no hay acaba -> StopEvent
 - Si hay llama a **handle_tool_calls**
 - **handle_tool_calls**:
 - Ejecuta la toolCall
 - Mete cada resultado en `tool_msgs`
 - Mete `tool_msgs` en `memory`
 - Vuelve a llamar al **handle_tool_calls** (ahora tiene más info en `memory` para responder)

a23-llamaindex-workflow-ToolCalls.py

Estrategias básicas

- Para mejorar nuestra aplicación LLMs
 - **Modelo:** utiliza un modelo de lenguaje con buen rendimiento y precisión
 - Pueden existir modelos entrados en dominios especializados (código, medicina, etc)
 - *Fine-tuning , adaptar a dominios o a tipos de respuesta
 - **Prompt Engineering:** refinar y mejorar los prompts propios o de los agentes
 - Inyección de rol actúa como...
 - **Chain-of-thought prompting** paso a paso, primero... luego..
 - **Few-shot prompting** (con ejemplos previos), de entrada y salida
 - **CLEAR** estructurar el contexto, limitación, objetivos, acciones/tarea y formato respuesta
 - Considera el uso de **Agentes** y **Herramientas**
 - Sus flujos y reglas pueden mejorar las salidas básicas de los LLMS
 - Refinamiento por interacción
 - Etc.
 - Uso de **Contexto/memoria**, para almacenar datos relevantes o históricos

Estrategias básicas

- Para mejorar nuestra aplicación LLMs
 - **Ensables:** utilizar varios modelos / agentes “votar” la respuesta
 - **Multiagente**
 - Utiliza agentes con diferentes roles “investigador”, “validador”, “sintetizador”
 - **Optimizar el costo**
 - Combinación de modelos pequeños y grandes
 - Utilización de cache
 - Define KPIs y evalúa las respuestas
 - Calidad, precisión, fluidez, utilidad, velocidad.