GESTIRE I PROPRI DOCUMENTI AZIENDALI CON LA RETRIEVAL-AUGMENTED GENERATION

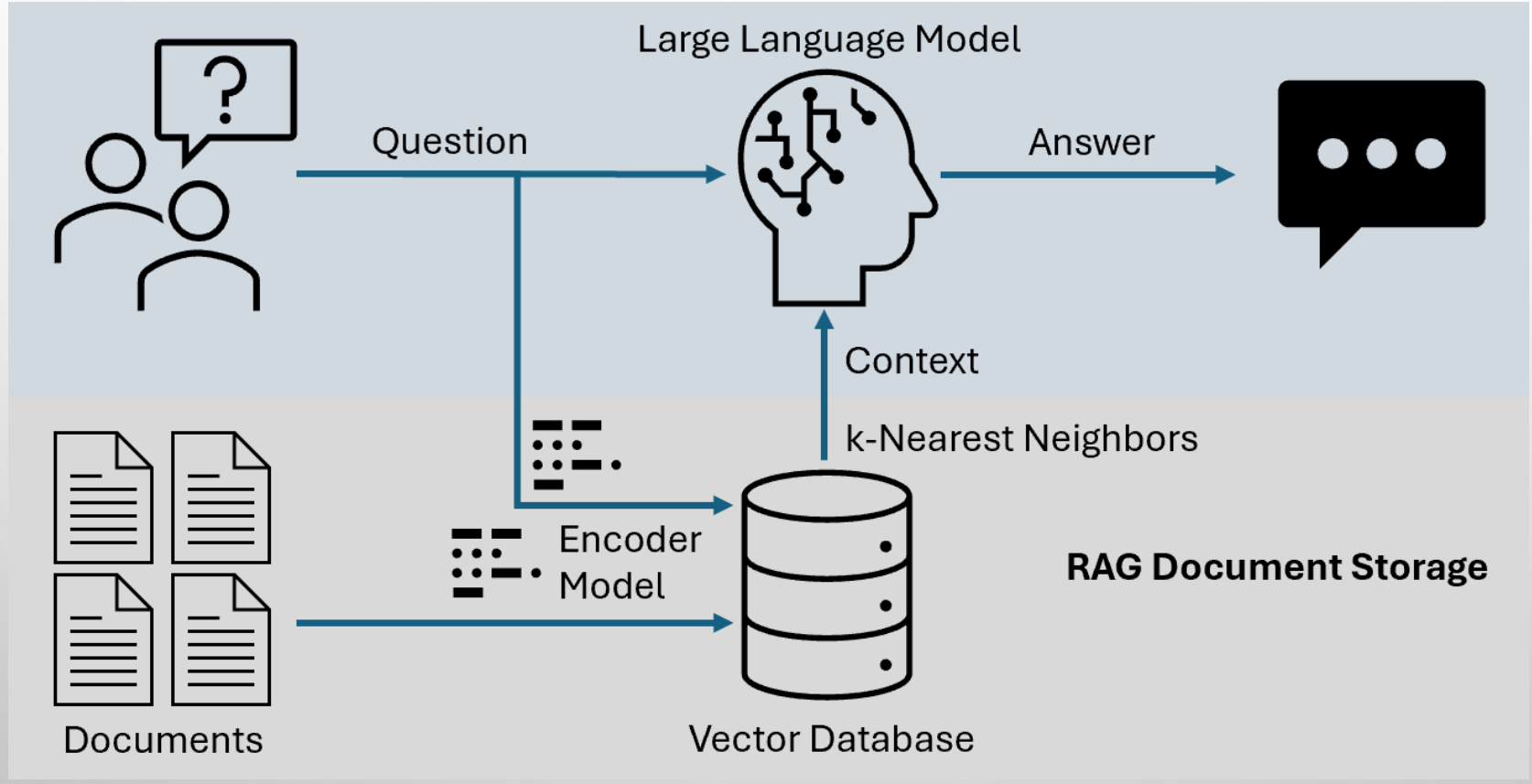
SVILUPPARE UN ECOSISTEMA IA COORDINATO DA LANGCHAIN PER GESTIRE EFFICACEMENTE INFORMAZIONI NON STRUTTURATE

Dr. Carlo Alberto Bentivoglio – UNIMC

RETRIEVAL-AUGMENTED GENERATION (RAG)

- LA RETRIEVAL-AUGMENTED GENERATION (**RAG**) È IL PROCESSO CHE PERMETTE AD UN LLM DI GENERARE RISPOSTE FACENDO RIFERIMENTO A UNA BASE DI CONOSCENZA AUTOREVOLE PIUTTOSTO CHE ALLE SUE FONTI DI DATI USATE DURANTE L'ADDESTRAMENTO.
- IN QUESTO MODO SI PUÒ UTILIZZARE L'ABILITÀ LINGUISTICA DEL MODELLO IN FUNZIONE DEI NOSTRI DOCUMENTI AZIENDALI PER OTTENERE RISPOSTE ALTAMENTE CONTESTUALIZZATE
- A TALE SCOPO, È NECESSARIO COSTRUIRE UN ECOSISTEMA PER GESTIRE L'INTERO PROCESSO COME SE FOSSE UN FLUSSO DI PROGRAMMAZIONE UTILIZZANDO IL FRAMEWORK **LANGCHAIN**

L'ECOSISTEMA RAG



ELEMENTI DELL'ECOSISTEMA

- **DATABASE VETTORIALE:** INDICIZZA I DOCUMENTI EFFETTUANDO L'EMBEDDING DEL DOCUMENTO E, IN BASE ALLA QUERY DELL'UTENTE SELEZIONA I DOCUMENTI PIÙ PERTINENTI
- **LLM:** CREA LA RISPOSTA UTILIZZANDO LA QUERY DELL'UTENTE E I DOCUMENTI SELEZIONATI DAL DB VETTORIALE COME CONTESTO ESCLUSIVO PER RISPONDERE
- **FRAMEWORK:** PUÒ ESSERE UTILE UTILIZZARE UN FRAMEWORK CHE GESTISCA IN MANIERA UNIFORME L'INTERA CATENA DALL'INPUT DELL'UTENTE ALLA GENERAZIONE DELLA RISPOSTA SOPRATTUTTO PER QUANTO RIGUARDA FONTI DATI ETEROGENEE

INSTALLARE L'AMBIENTE DI BASE

- REQUISITI 'MINIMI' DI SISTEMA
  - 32 GB DI RAM
  - SCHEDA GRAFICA NVIDIA
- INSTALLARE OLLAMA
  - HTTPS://OLLAMA.COM/DOWNLOAD
- INSTALLARE PYTHON
  - HTTPS://WWW.PYTHON.ORG/DOWNLOADS/RELEASE/PYTHON-3132/
- INSTALLARE PIP
  - CURL HTTPS://BOOTSTRAP.PYPA.IO/GET-PIP.PY -O GET-PIP.PY
  - PY GET-PIP.PY
- INSTALLARE LANGCHAIN **IN FUNZIONE DI OLLAMA**
  - PIP INSTALL -U LANGCHAIN-OLLAMA

OLLAMA

- OLLAMA PERMETTE DI UTILIZZARE SUL PROPRIO COMPUTER GLI LLM OPENSOURCE PRESENTI SUL PROPRIO SITO
- VIENE ISTALLATO COME SERVIZIO E PUÒ ESSERE ACCEDUTO VIA API O CON UN CLIENT A LINEA DI COMANDO
- ALCUNI COMANDI
  - OLLAMA RUN DEEPSEEK-R1:8B (CARICA ED ESEGUE UN LLM)
  - OLLAMA PULL DEEPSEEK-R1:8B (CARICA UN LLM)

OLLAMA

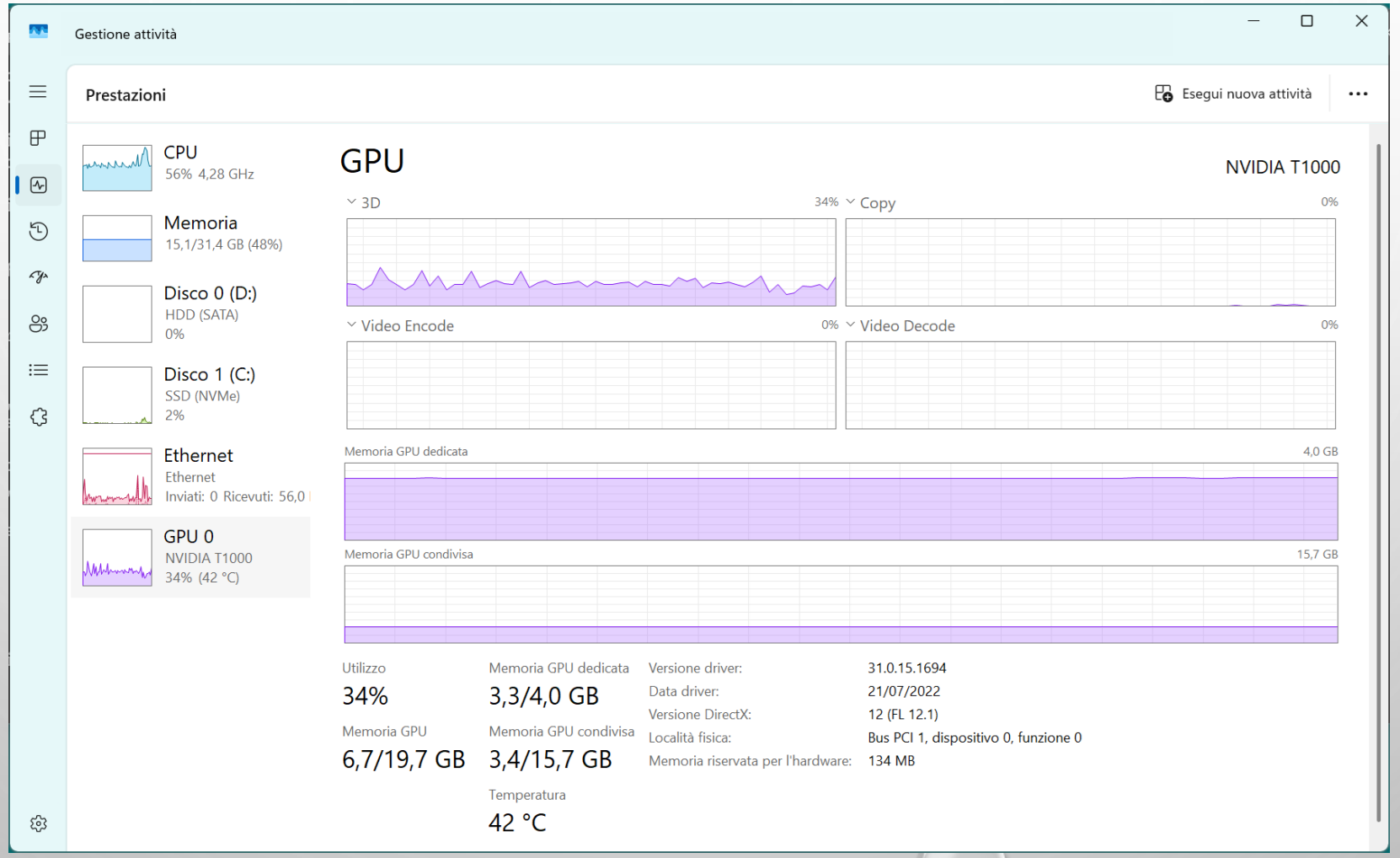
- OLLAMA LIST (ELENCA GLI LLM PRESENTI)

LLAMA3.2:3B	A80C4F17ACD5	2.0 GB	3 MINUTES AGO
NOMIC-EMBED-TEXT:LATEST	0A109F422B47	274 MB	4 WEEKS AGO
VAITON/MINERVA:LATEST	68A06AA7974D	3.1 GB	8 WEEKS AGO
DEEPSEEK-R1:8B	28F8FD6CDC67	4.9 GB	8 WEEKS AGO
DEEPSEEK-R1:1.5B	A42B25D8C10A	1.1 GB	8 WEEKS AGO

ALCUNE INFORMAZIONI SUL SERVER OLLAMA

- OLLAMA RISPONDE ALLA PORTA 11434
  - CURL -H "CONTENT-TYPE: APPLICATION/JSON" -D '{"MODEL": "DEEPSEEK-R1:8B","PROMPT": "WHO ARE YOU?"}' HTTP://LOCALHOST:11434/API/GENERATE
- EXPLORER %LOCALAPPDATA%\OLLAMA (FOLDER CONTENENTE LOG E MODELLI)
  - SERVER.LOG
  - APP.LOG
  - UPGRADE.LOG

IMPEGNO COMPUTAZIONALE



STRUTTURE DATI IN PYTHON

```
(1,2)
```

Tuple: liste di elementi immutabili

```
{"red","blue","green"}
elementi
```

Set: collezioni a cui posso aggiungere o togliere

```
{"model": "deepseek",
"prompt": "Who are you?"}
```

Dictionary: collezioni di coppie chiave:valore

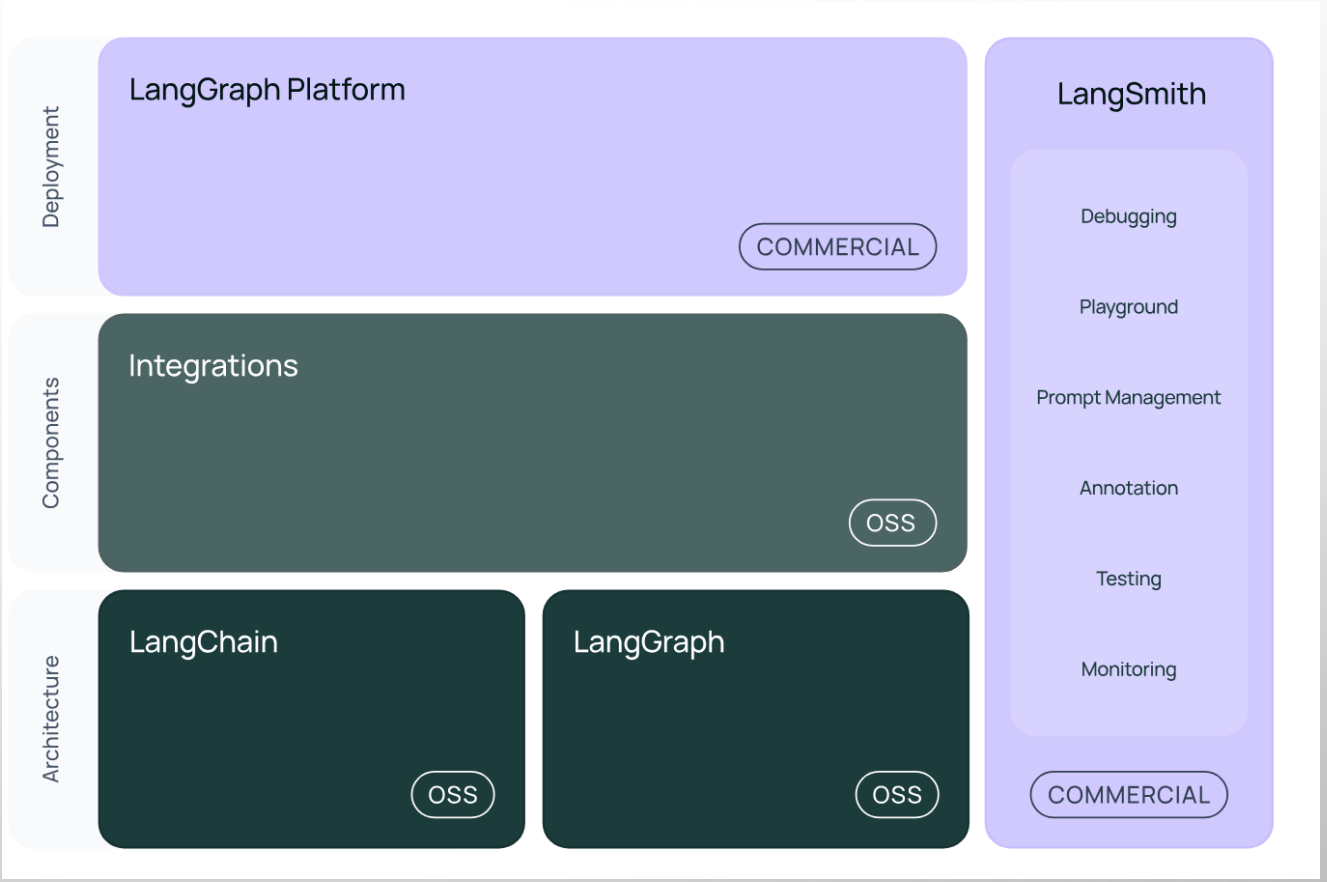
```
[1,2,3]
```

List: collezioni ordinate di elementi

LangChain

LANGCHAIN È COSTITUITO DA **SERVIZI DI BASE** E **INTEGRAZIONI CON PRODOTTI ESTERNI** SOTTO FORMA DI SOFTWARE OPEN SOURCE.

AL CONTRARIO, LE FUNZIONALITÀ A LIVELLO ENTERPRISE SONO A PAGAMENTO



NOTA BENE

IN LANGCHAIN SPESSO LO STESSO RISULTATO PUÒ ESSERE OTTENUTO UTILIZZANDO LIVELLI D'ASTRAZIONE DIVERSI. AD ESEMPIO NELL'ORCHESTRARE I SERVIZI SI HANNO ALMENO TRE LIVELLI:

- SEQUENZE DI CHIAMATE DIRETTE A OGGETTI
- CATENE
- GRAFI

LO SCOPO DI USARE METODOLOGIE O LIVELLI DI ASTRAZIONE PIÙ ALTI È UTILE, SOPRATTUTTO, PER RENDERE IL CODICE PIÙ MANUTENIBILE O SCRIVERE MENO CODICE.

IL PRIMO PROMPT VERSO OLLAMA

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_ollama.llms import OllamaLLM

template = """Question: {question}

Answer: Let's think step by step."""

prompt = ChatPromptTemplate.from_template(template)

model = OllamaLLM(model="deepseek-r1:8b")

chain = prompt | model

a = chain.invoke({"question": "Where was born Matteo Ricci, and where he died?"})

print(a)
```

Il metodo **invoke** sulla catena (che a sua volta richiama invoke sul modello) permette di interrogare l'LLM specificando il valore dei parametri definiti nel template mediante un oggetto di tipo dictionary

Il **template** ci aiuta ad adattare la domanda dell'utente al modo in cui vogliamo porre la domanda al LLM suggerendogli il modo e lo stile di risposta (prompt engineering)

chatPromptTemplate prepara il **prompt**, a partire da un template, che sarà inviato al LLM

OllamaLMM imposta il **modello** di LLM a cui sarà inviato il prompt ed i parametri da utilizzare per quel modello (ad esempio la temperatura)

Il flusso di informazione verso LLM è codificato con una **chain** che lega il prompt verso il model. Notate il significato letterale del simbolo | (pipe) per legare i due elementi

Per approfondire i vari concetti si consiglia:
https://python.langchain.com/docs/concepts/

LA RISPOSTA… DI DEEPSEEK (DISTILLED)

PURTROPPO IL RISULTATO È IL SEGUENTE (SI NOTI LA PARTE DI REASONING TRA I TAG <THINK>)

<THINK> OKAY, SO I'M TRYING TO FIGURE OUT WHERE MATTEO RICCI WAS BORN AND WHERE HE DIED. I REMEMBER HEARING HIS NAME IN THE CONTEXT OF JESUITS AND MISSIONS IN CHINA, BUT I'M NOT EXACTLY SURE ABOUT THE DETAILS. LET ME START BY RECALLING WHAT I KNOW. MATTEO RICCI IS A WELL-KNOWN ITALIAN MISSIONARY WHO PLAYED A SIGNIFICANT ROLE IN LATE MING DYNASTY CHINA. HE WAS PART OF THE JESUIT ORDER, WHICH WAS ACTIVE DURING THE 17TH CENTURY. THE JESUITS WERE INVOLVED IN VARIOUS MISSIONARY ACTIVITIES, OFTEN UNDER ORDERS FROM THE CATHOLIC CHURCH TO SPREAD CHRISTIANITY. I THINK RICCI WAS BORN SOMEWHERE IN ITALY. ITALY HAS SEVERAL REGIONS, BUT I'M NOT SURE EXACTLY WHICH CITY. ROME IS A BIG PLACE FOR RELIGIOUS FIGURES, SO MAYBE HE WAS BORN THERE? BUT I'M NOT CERTAIN. I SHOULD CONSIDER OTHER POSSIBILITIES TOO. MAYBE HE WAS FROM ANOTHER CITY IN ITALY, LIKE FLORENCE OR VENICE. WAIT, I THINK I'VE HEARD THAT HE WAS FROM TUSCANY, BUT I'M NOT ENTIRELY SURE. NOW, ABOUT HIS DEATH. HE DIED DURING THE EARLY YEARS OF THE KANGXI REIGN, WHICH WAS AROUND 1662. SO HE MUST HAVE BEEN ALIVE DURING THE LATE MING PERIOD AND DIED AS THE QING WERE TAKING OVER. SINCE HE WAS A MISSIONARY, HE PROBABLY TRAVELED TO CHINA THROUGH MACAU, WHICH WAS A PORTUGUESE COLONY AT THE TIME. MACAU IS IN PRESENT-DAY CHINA BUT WAS A SEPARATE TERRITORY THEN.AS FOR WHERE HE DIED, I BELIEVE HE PASSED AWAY IN BEIJING BECAUSE THAT'S WHERE THE IMPERIAL COURT WAS LOCATED, AND HE HAD ACCESS TO HIGH-RANKING OFFICIALS. BUT I'M NOT ENTIRELY SURE IF HE DIED THERE OR SOMEWHERE ELSE. MAYBE HE WAS BURIED IN A CHURCH OR A SPECIFIC LOCATION IN BEIJING. WAIT, LET ME THINK ABOUT HIS CONTRIBUTIONS. RICCI WROTE THE "TAO OF CONFUCIUS," WHICH WAS AN ATTEMPT TO PRESENT CONFUCIAN PHILOSOPHY TO THE CATHOLIC CHURCH. HE ALSO PROMOTED SCIENTIFIC AND CULTURAL EXCHANGES BETWEEN CHINA AND EUROPE. HIS WORK LAID THE GROUNDWORK FOR MORE OPEN INTERACTIONS BETWEEN THE TWO REGIONS. I'M TRYING TO REMEMBER IF THERE'S ANY SPECIFIC EVENT THAT MARKS HIS DEATH. HE MIGHT HAVE DIED DURING A SPECIFIC MISSION OR AFTER A PARTICULAR ACHIEVEMENT. I THINK HE WAS RESPECTED BY THE CHINESE AUTHORITIES, SO HIS DEATH MIGHT HAVE BEEN NOTED IN HISTORICAL RECORDS. ANOTHER ANGLE: LOOKING UP WHERE OTHER JESUITS FROM THAT TIME WERE BORN AND DIED COULD HELP. FOR EXAMPLE, OTHER JESUIT MISSIONARIES LIKE XAVIER OR RODRIGUES HAD THEIR ORIGINS IN EUROPE, OFTEN SPAIN OR PORTUGAL BECAUSE THOSE WERE MAJOR AREAS FOR MISSIONARY WORK. RICCI WAS ITALIAN, SO MAYBE HE CAME FROM A DIFFERENT REGION BUT SIMILAR CIRCUMSTANCES. I SHOULD ALSO CONSIDER THE POLITICAL CLIMATE AT THE TIME. THE LATE MING PERIOD WAS UNSTABLE, LEADING UP TO THE MANCHU CONQUEST. RICCI MIGHT HAVE BEEN INVOLVED IN DIPLOMATIC EFFORTS OR DEALING WITH POLITICAL CHANGES AS THE QING TOOK OVER. IN TERMS OF SOURCES, I CAN RECALL THAT HIS LIFE IS DOCUMENTED IN HISTORICAL TEXTS ABOUT THE INTERACTIONS BETWEEN CHINA AND EUROPE DURING THE 17TH CENTURY. MAYBE LOOKING INTO BIOGRAPHIES OR PRIMARY ACCOUNTS FROM JESUIT LETTERS WOULD GIVE MORE PRECISE INFORMATION. I'M A BIT UNSURE IF HE WAS BORN IN ROME OR ANOTHER CITY, BUT CONSIDERING HIS RELIGIOUS BACKGROUND AS A JESUIT, IT'S LIKELY HE WAS BORN IN A PROMINENT CATHOLIC AREA IN ITALY. ROME IS A BIG HUB FOR THE CATHOLIC CHURCH, SO THAT MAKES SENSE. PUTTING THIS ALL TOGETHER, I THINK MATTEO RICCI WAS BORN IN ITALY, PROBABLY IN TUSCANY OR ROME, AND DIED IN BEIJING DURING 1662 AS PART OF THE TRANSITION FROM MING TO QING RULE.**</THINK>**

MATTEO RICCI WAS AN ITALIAN MISSIONARY BORN IN PISTOIA, A CITY IN TUSCANY, ITALY. HE DIED ON AUGUST 21, 1662, IN BEIJING, CHINA, DURING THE EARLY YEARS OF THE KANGXI EMPEROR'S REIGN. HIS DEATH OCCURRED AS THE MANCHU FORCES WERE CONSOLIDATING THEIR RULE, TRANSITIONING FROM THE MING TO THE QING DYNASTY. RICCI IS NOTED FOR HIS SIGNIFICANT CONTRIBUTIONS TO CULTURAL AND SCIENTIFIC EXCHANGES BETWEEN CHINA AND EUROPE, PARTICULARLY THROUGH HIS EFFORTS IN TRANSLATING CONFUCIAN TEXTS AND PROMOTING MUTUAL UNDERSTANDING.

TIPOLOGIE DI CLASSI COINVOLTE

Le tipologie di classi coinvolte nella formazione delle catene sono le seguenti:

- **PromptTemplate**: generano il prompt da mandare all'LLM in base alle richieste dell'utente e ai contesti disponibili
- **Retriever**: in base ad una richiesta dell'utente raccolgono dati dall'esterno per fornire un contesto dal quale trarre maggiori informazioni per rispondere all'utente (dal web, da un db vettoriale, …)
- **LLM**: generano la risposta dell'LLM
- **OutputParser**: formattano l'output in maniera oppotuna (testo strutturato e non)

Queste classi implementano l'intefaccia **Runnable** che permette di:

- collegare l'output di uno con l'output dell'altro mediante l'operatore | (pipe)
- Invocare, in modo uniforme rispetto alla classe, una serie di metodi per ottenere il risultato secondo modalità diverse (tutto in una volta, in streaming, sincrono/asincrono …)

Per approfondire:
https://python.langchain.com/docs/concepts/runnables/

EFETTUARE LO STREAMING DELLA RISPOSTA

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_ollama.llms import OllamaLLM

from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
from langchain.callbacks.manager import CallbackManager

callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])

template = """Question: {question}

Answer: Let's think step by step."""

prompt = ChatPromptTemplate.from_template(template)

model = OllamaLLM(model="deepseek-r1:8b", callbacks=callback_manager)

chain = prompt | model

q = input(": ")

a = chain.invoke({"question": q})

print(a)
```

Per approfondire:
https://python.langchain.com/docs/how_to/streaming_llm/
https://github.com/langchain-ai/langchain/issues/13333

Nel caso precedente la chiamata sincrona attendenva diversi minuti prima di dare, tutta in una volta, la risposta dell'LLM

In questo caso, effettuando il subclassing dello StdOut, intercettiamo lo stream di token che vengono generate una alla volta dall'LLM e le stampiamo man mano che arrivano

UN ALTRO MODO DI FARE STREAMING -1

pip install streamlit

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_ollama.llms import OllamaLLM

import streamlit as st

template = """Question: {question}
Answer: Let's think step by step."""

def call_llm(q: str):

    prompt = ChatPromptTemplate.from_template(template)

    model = OllamaLLM(model="deepseek-r1:8b")

    chain = prompt | model

    a = chain.stream({"question": q})

    thinking = True

    for chunk in a:
        if not thinking:
            yield chunk
        if chunk == "<think>":
            thinking = True
        elif chunk == "</think>":
            thinking = False
```

Un altro modo di fare streaming consiste nel utilizzare il metodo stream sulla chain, per poi iterare sul suo risultato e ottenere i token volta per volta.

Attraverso **yeld**, anziché return, è possibile mantenere lo stato della computazione tra una chiamata e l'altra alla medesima funzione.
In questo modo è possibile mandare via i token man mano che arrivano dal flusso di streaming dell'LLM.

Le funzioni che utilizzano yeld vengono chiamate in python funzioni generative

UN ALTRO MODO DI FARE STREAMING -2

```
if __name__ == "__main__":

  st.header("OLLAMA Question Answer")
  qu = st.text_area("Ask a question:")
  ask = st.button(
        " Ask",
  )

  if qu and ask:
    response = call_llm(q=qu)
    st.write_stream(response)
```

Per lanciare l'applicazione dalla sua directory

streamlit run chat2.py

Per approfondire:
https://streamlit.io/

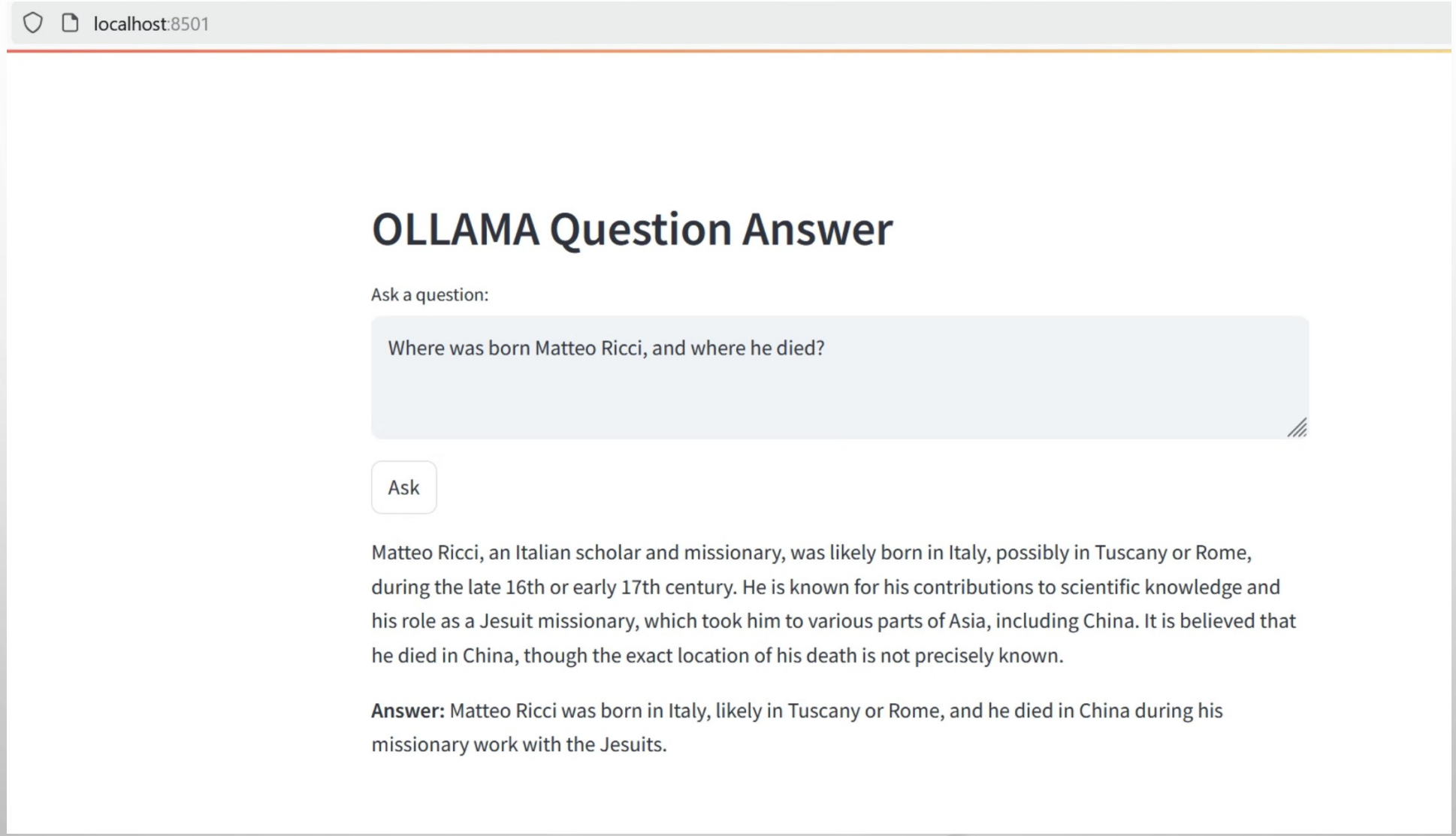
Per avere velocemente un riscontro di come possa essere interrogata via web la nostra applicazione si può usare l'ambiente streamlit

Questo ingloba l'applicazione all'interno di un server che funge da interfaccia web rispondendo alla porta 8501.

In questo modo si può definire il comportamento di bottoni, caselle di input e altro direttamente nel codice dell'applicazione

Si noti il metodo write_stream che prende in input il risultato della funzione generatrice (un iteratore) e stampa via via i token in maniera del tutto trasparente facendo scorrere l'iteratore.

IL RISULTATO…



USARE INFORMAZIONI DI CONTESTO DA WIKIPEDIA - 1

- pip install -qU langchain_community wikipedia

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_ollama.llms import OllamaLLMfrom

from langchain_community.retrievers import WikipediaRetriever

def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

model = OllamaLLM(model="deepseek-r1:8b")

retriever = WikipediaRetriever()
docs = retriever.invoke("Matteo Ricci")

prompt = ChatPromptTemplate.from_template(
    """
    Answer the question based only on the context provided.
    Context: {context}
    Question: {question}
    """
)
```

La funzione **format_docs** semplicemente unisce il contenuto di più documenti in uno solo

Si imposta il **retriver** ad un apposito oggetto che sa come consultare Wikipedia. Il metodo invoke in questo caso ricerca la voce desiderata

Il prompt viene inizializzato con un template che permette di specificare un contesto prima della domanda vera e propria

Per approfondire:
https://python.langchain.com/docs/integrations/retrievers/wikipedia/

USARE INFORMAZIONI DI CONTESTO DA WIKIPEDIA - 2

```
chain = (
    {"context": retriever | format_docs, "question":
RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)

a = chain.invoke("Where was born Matteo Ricci, and
where he died?")

print(a)
```

Il primo anello della catena rappresenta l'argomento del prompt, ovvero una dictionary in cui context è istanziato da una 'sottocatena' e question dalla funzione RunnablePassthrough().

In particolare, l'output del retriver diventa l'input di format_docs che unisce i documenti in un'unica variabile che fungerà da valore per l'argomento context.

L'utilizzo del RunnablePassthrough() permette di dilazionare al momento dell'invocazione della catena il contenuto della variabile question

Per approfondire:
https://python.langchain.com/docs/integrations/retrievers/wikipedia/

USARE INFORMAZIONI DI CONTESTO DA WIKIPEDIA - 3

Al contrario del caso senza informazioni specifiche, risultato è il seguente:

<think>Okay, so I'm trying to figure out where Matteo Ricci was born and where he died. From the information given in the text, it mentions that Ricci arrived in Macau on August 6, 1578, after spending some time in Goa, India. The question is about his birthplace and death location First, I know that Matteo Ricci was an Italian Jesuit missionary who played a significant role in bringing European knowledge to China during the late Ming dynasty. He's known for translating scientific works from Latin into Chinese, which helped bridge the gap between Chinese and Western learning at the time. The text says he arrived in Macau after spending some time in Goa, so I assume he was there before. But where was he born? Since Ricci was an Italian, his birthplace is likely in Italy. The Jesuits had missions in various parts of Europe, including Italy. However, without specific information about his family background or exact place of birth, it's a bit tricky. Looking at the references provided, there's a mention of "Matteo Ricci" and some sources like Ignacio Ramos Riera's work on Diego de Pantoja. I recall that Matteo Ricci was born in 1552 in Macerata, Italy. That seems plausible because Macerata is a city in Marche, near the Adriatic Sea. It makes sense given the historical context of the Jesuit missions in Italy and their expansion into Asia. As for his death, the text mentions that he died on October 21, 1610, in Beijing. He was buried in the Church of St. Peter ad Aram in Beijing, which is now a famous site associated with him. So, putting it all together, Ricci was born in Macerata, Italy, and died in Beijing, China. I should make sure I'm not confusing him with other Jesuits. Diego de Pantoja is mentioned as accompanying Ricci to Beijing but died later. So Ricci's death is indeed in Beijing, while de Pantoja was expelled and died in Macau. **</think>**

Matteo Ricci was born on April 24, 1552, in Macerata, Italy. He passed away on October 21, 1610, in Beijing, China, where he was buried in the Church of St. Peter ad Aram.

Per approfondire:
https://python.langchain.com/docs/integrations/retrievers/wikipedia/

RIFLESSIONE SUL PROMPT CHE INVIAMO

QUANDO DEFINIAMO DEI TEMPLATE, I CUI CAMPI APPOSITAMENTE VALORIZZATI NEL CORSO DELL'ELABORAZIONE, DIVENTANO DEI PROMPT STIAMO CONVERSANDO CON LLM IN **LINGUAGGIO NATURALE**.

CIÒ CHE CI ASPETTIAMO (NEL MODO, NEL CONTENUTO, NELLO STILE) DELLA RISPOSTA **NON È STATO ESPLICITAMENTE PROGRAMMATO** MA È **EMERSO** NEL CORSO DELL'ADDESTRAMENTO

AD ESEMPIO, **NESSUNO HA PROGRAMMATO** IL LARGE LANGUAGE MODEL IN MODO CHE RICONOSCA LA PAROLA CONTEXT E SI BASI ESCLUSIVAMENTE SU QUELLO CHE SEGUE PER GENERARE LA RISPOSTA

DATABASE VETTORIALI

I DATABASE VETTORIALI PERMETTONO DI FARE L'EMBEDDING DI INTERI DOCUMENTI.

IN PRATICA, AD UN CERTO DOCUMENTO VIENE ASSOCIATO UN VETTORE IN UNO SPAZIO MULTIDIMENSIONALE COME AVVIENE CON NEL PRIMO STRATO DEGLI LLM.

IN QUESTO MODO, DATO UN CERTO PROMPT IL SISTEMA TROVA I DOCUMENTI PIÙ PERTINENTI IN MODO CHE IL LORO CONTENUTO POSSA FUNGERE DA CONTESTO ESCLUSIVO SUL QUALE LLM GENERERÀ LA RISPOSTA.

INOLTRE, PER RENDERE PIÙ FOCALIZZATA LA RICERCA È POSSIBILE ASSOCIARE AL DOCUMENTO DI CUI SI FA L'EMBEDDING ANCHE DEI METADATI.

Chroma

Poiché LLM e DB devono avere lo stesso strato di embedding, per prima cosa dobbiamo caricare l'embedding che useremo con Chroma su ollama e che useremo per fare l'embedding dei documenti su db

ollama pull nomic-embed-text

Per semplicità istallaremo la versione non persistente (ovvero che memorizza in ram i dati e non sul disco)

pip install chromadb

Installiamo le librerie per langchain

pip install --upgrade --quiet lark

pip install --upgrade --quiet langchain-chroma

Per approfondimenti:
https://docs.trychroma.com/docs/overview/getting-started
https://python.langchain.com/docs/integrations/providers/chroma/

RICERCA SEMANTICA SU UN DOCUMENTO -1

pip install pypdf

```
from langchain_community.document_loaders import PyPDFLoader

file_path = "d:/AI/data/nke-10k-2023.pdf"
loader = PyPDFLoader(file_path)
docs = loader.load()

from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200, add_start_index=True
)
all_splits = text_splitter.split_documents(docs)

from langchain_ollama import OllamaEmbeddings
embeddings = OllamaEmbeddings(model="nomic-embed-text")

from langchain_chroma import Chroma

vector_store = Chroma(
    collection_name="example_collection",
    embedding_function=embeddings,
)

ids = vector_store.add_documents(documents=all_splits)
```

Carico il pdf e lo suddivido in tanti chunk tra loro sovrapposti in modo da mantenere meglio il contesto

Utilizzo Ollama per fare l'embedding dei documenti

Utilizzo, in memoria, il db vettoriale chroma per memorizzare i documenti, i metadati e la corrispondente rappresentazione vettoriale (embedding)

RICERCA SEMANTICA SU UN DOCUMENTO -2

```
results = vector_store.similarity_search_with_score(
    "How many distribution centers does Nike have in the US?"
)

doc, score = results[0]

print(f"Score: {score}\n")
```

Score: 0.3253529965877533

```
print(doc)
```

page_content='direct to consumer operations sell products through the following number of retail stores in the United States:
U.S. RETAIL STORES NUMBER
NIKE Brand factory stores 213
NIKE Brand in-line stores (including employee-only stores) 74
Converse stores (including factory stores) 82
TOTAL 369
In the United States, NIKE has eight significant distribution centers. Refer to Item 2. Properties for further information.
2023 FORM 10-K 2'

metadata={'author': 'EDGAR Online, a division of Donnelley Financial Solutions', 'creationdate': '2023-07-20T16:22:00-04:00', 'creator': 'EDGAR Filing HTML Converter', 'keywords': '0000320187-23-000039; ; 10-K', 'moddate': '2023-07-20T16:22:08-04:00', 'page': 4, 'page_label': '5', 'producer': 'EDGRpdf Service w/ EO.Pdf 22.0.40.0', 'source': 'd:/AI/data/nke-10k-2023.pdf', 'start_index': 3125, 'subject': 'Form 10-K filed on 2023-07-20 for the period ending 2023-05-31', 'title': '0000320187-23-000039', 'total_pages': 107}

Interrogo il db con una richiesta in linguaggio naturale e ottengo una lista ordinata per rilevanza dei documenti che il sistema giudica pertinenti alla richiesta

Lo score in questo caso è la distanza coseno ovvero il complemento della similarità coseno (nel primo caso, più è basso il valore e meglio è)

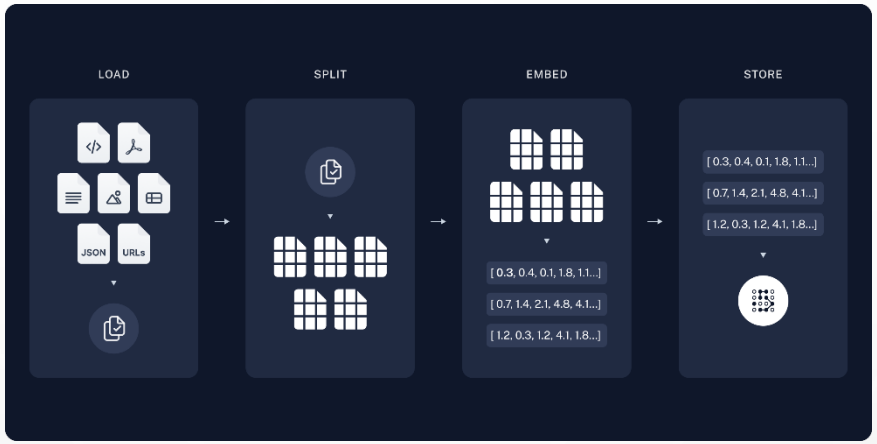
RICERCA SEMANTICA SU UN DOCUMENTO -3

```
retriever = vector_store.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"k":1,"score_threshold":0.2},
)

retriever.invoke("How many distribution centers does
Nike have in the US?")
```

Per poter utilizzare la ricerca su un db vettoriale all'interno di una **chain** è necessario 'impacchettare' l'interfaccia verso il db all'interno di un oggetto **retriever** che espone tutti i metodi necessari all'ecosistema.

Per approfondire:
https://python.langchain.com/docs/tutorials/retrievers/

RAG MINIMALE - INDEXING



pip install langgraph

```
import bs4
from langchain_community.document_loaders import WebBaseLoader

bs4_strainer = bs4.SoupStrainer(class_=("post-title", "post-header", "post-
content"))
loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
    bs_kwargs={"parse_only": bs4_strainer},
)
docs = loader.load()
```

Il retriever WebBaseLoader utilizza bs4 per fare il parsing della pagina HTML e ottenere un testo significativo

Per approfondire

https://python.langchain.com/docs/tutorials/rag/

RAG MINIMALE - INDEXING

```
from langchain_ollama.llms import OllamaLLM
llm = OllamaLLM(model="deepseek-r1:8b")

from langchain_ollama import OllamaEmbeddings
embeddings = OllamaEmbeddings(model="nomic-embed-text")

from langchain_chroma import Chroma

vector_store = Chroma(
    collection_name="example_collection",
    embedding_function=embeddings,
)
```

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,  # chunk size (characters)
    chunk_overlap=200,  # chunk overlap (characters)
    add_start_index=True,  # track index in original document
)
all_splits = text_splitter.split_documents(docs)

ids = vector_store.add_documents(documents=all_splits)
```

Il documento preso da internet viene spezzettato in tanti chunk più piccoli e caricato nel database vettoriale

RAG MINIMALE - LANGRAPH

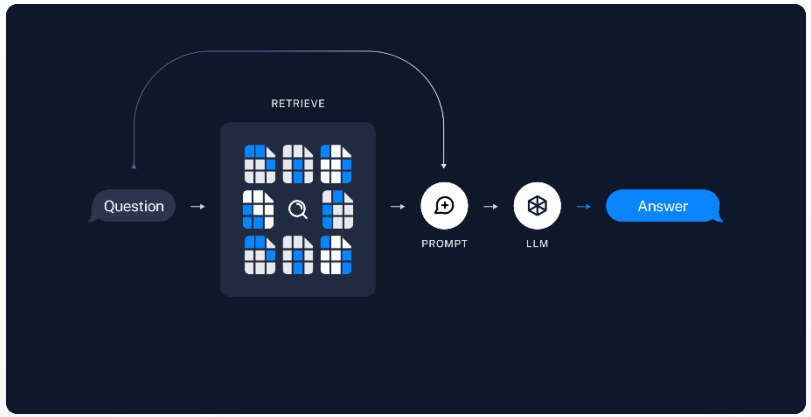
- **Langraph** è una 'tecnologia' che permette di orchestrare interazioni con i vari elementi dell'ecosistema.

- Ogni possibile azione è modellata come il nodo di un **grafo**. Il nodo successivo da eseguire è scelto in base allo stato del sistema.

- Le **chain** sono un caso particolare di grafi sequenziali in cui i nodi sono i vari oggetti presenti nella catena e sono eseguiti uno dopo l'altro in maniera esclusivamente deterministica

- Per costruire un grafo è necessario, quindi, individuare tre elementi fondamentali:
  - a) lo **stato** dell'applicazione
  - b) i **nodi** dell'applicazione (i passi di una procedura tradizionale)
  - c) il **flusso di controllo** dell'applicazione (l'ordine dei passi)

RAG MINIMALE – PROMPT TEMPLATE




```
from langchain import hub
prompt = hub.pull("rlm/rag-prompt")

example_messages = prompt.invoke(
    {"context": "(context goes here)", "question": "(question goes here)"}
).to_messages()
```

You are an assistant for question-answering tasks. Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know. Use three sentences maximum and keep the answer concise.
Question: (question goes here)
Context: (context goes here)
Answer:

Definiamo un template per il prompt, a partire da un 'semilavorato' fornito dal framework, del tipo:

Definiamo una funzione retrieve per ottenere i doc di contesto
La funzione retrieve sarà uno dei nodi del grafo che andremo a implementare

RAG MINIMALE – RETRIVE AND GENERATE

```
from langchain_core.documents import Document
from typing_extensions import List, TypedDict
from langgraph.graph import START, StateGraph

class State(TypedDict):
    question: str
    context: List[Document]
    answer: str

def retrieve(state: State):
    retrieved_docs = vector_store.similarity_search(state["question"])
    return {"context": retrieved_docs}

def generate(state: State):
    docs_content = "\n\n".join(doc.page_content for doc in state["context"])
    messages = prompt.invoke({"question": state["question"], "context": docs_content})
    response = llm.invoke(messages)
    return {"answer": response}

graph_builder = StateGraph(State).add_sequence([retrieve, generate])
graph_builder.add_edge(START, "retrieve")
graph = graph_builder.compile()
```

Definisco lo stato

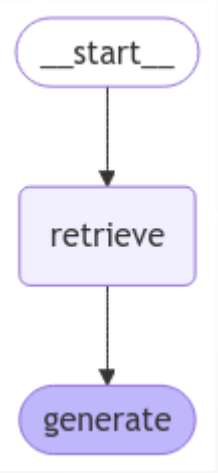
Definisco i due nodi:
retrieve
generate

Costruisco il grafo aggiungendo all'inizio della sequenza il nodo START

Notare la compilazione finale

RAG MINIMALE – GENERARE LA RISPOSTA

```
img = graph.get_graph().draw_mermaid_png()

with open(r"d:\fsm.png", "wb") as png:
            png.write(img)
```



Per vedere cosa abbiamo costruito è possibile far disegnare il grafo come immagine png

```
result = graph.invoke({"question": "What is
Task Decomposition?"})

print(f'Context: {result["context"]}\n\n')
print(f'Answer: {result["answer"]}')
```

A questo punto effettuo la richiesta sul grafo che ho appena costruito

Context: [**Document**(id='40e531b3-18ff-483e-bd78-8980f70c9c96', metadata={'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 1585}, **page_content**='Fig. 1. Overview of a LLM-powered autonomous agent system.\nComponent One: Planning#\nA complicated task usually involves many steps. An agent needs to know what they are and plan ahead.\nTask Decomposition#\nChain of thought (CoT; Wei et al. 2022) has become a standard prompting technique for enhancing model performance on complex tasks. The model is instructed to “think step by step” to utilize more test-time computation to decompose hard tasks into smaller and simpler steps. CoT transforms big tasks into multiple manageable tasks and shed lights into an interpretation of the model’s thinking process.'),…]

Answer: <think>
Okay, so I'm trying to figure out what Task Decomposition is based on the context provided. Let me read through the context carefully.
The first part talks about how a complicated task usually involves many steps and that an agent needs to know these steps and plan ahead. So, it's like breaking down big tasks into smaller, manageable parts. Then there's mention of Chain of Thought (CoT) being a standard technique to enhance model performance on complex tasks. CoT makes the model think step by step, using more computation time at test stage. It transforms big tasks into multiple simpler ones and gives insight into how the model thinks. …</think>

Task decomposition is the method of breaking down a complex task into simpler, sequential steps. It uses techniques like Chain of Thought (CoT) to enhance model reasoning, enabling them to tackle complicated issues by thinking step-by-step. The AI can parse user input into structured tasks with dependencies and use appropriate models to execute each part effectively.

DECORATORI

Il **decoratore** (in inglese **decorator**) è un pattern OOP che consente di estendere e modificare il comportamento di funzioni e classi senza doverne alterarne il codice sorgente.

In python, il pattern è implementato con *una funzione (o una classe)* che:

- prende come parametro un'altra funzione
- aggiunge delle funzionalità
- restituisce un'altra funzione senza alterare il codice sorgente della funzione passata come parametro

```
def funzione_decoratore(funzione_parametro):
    def wrapper():
        print("... codice da eseguire prima di 'funzione_parametro' ...")
        funzione_parametro()
        print("... codice da eseguire dopo di 'funzione_parametro' ...")
    return wrapper

def mia_funzione():
    print("Hello World!")

@funzione_decoratore
def mia_funzione():
    print("Hello World!")

mia_funzione()
```

```
... codice da eseguire prima di funzione_parametro ...
hello world!
... codice da eseguire dopo di funzione_parametro ...
```

Fonte:
https://www.programmareinpython.it/video-corso-python-intermedio/12-i-decoratori/

UTILIZZARE FUNZIONI (TOOL) - 1

```
import streamlit as st
from langchain_core.messages import HumanMessage,
ToolMessage
from langchain_core.tools import tool
from langchain_ollama import ChatOllama

messages = []
prompt = st.text_input("Enter your prompt")

@tool(parse_docstring=True)
def get_square(number: int) -> str:
    """Returns the square of a number, for example
when a customer asks "What is the square of 10?"

    Args:
        number: a number whose the user want to
calculate the square

    Note: View JSON Schema:
get_expert.args_schema.schema()

    Returns:
        str: the square of the number
    """

    return number*number

tools_list = {
    "get_square": get_square,
}
```

Alcuni LLM permettono l'utilizzo di funzioni definite dall'utente all'interno del processo di generazione della risposta.

Tali funzioni possono essere anche delle chiamate esterne permettendo all'LLM di operare come un agente intelligente nel cyberspazio.

Tali funzioni sono definite tool e vanno definite in modo che l'LLM capisca come usarle. Per renderne possibile l'utilizzo, si effettua un wrapping col decoratore @tool

L'elenco dei tool viene poi sottoposto all'LLM mediante una lista di tali funzioni

UTILIZZARE FUNZIONI (TOOL) - 2

```
if prompt:
    llm = ChatOllama(model="llama3.2:3b")
    llm_with_tools = llm.bind_tools(list(tools_list.values()))

    messages.append(HumanMessage(prompt))
    ai_response = llm_with_tools.invoke(messages)
    messages.append(ai_response)

    if not ai_response.tool_calls:
        with st.container(height=500, border=True):
            st.write(ai_response.content)
            sys.exit()

    for tool_call in ai_response.tool_calls:
        selected_tool = tools_list.get(tool_call["name"].lower())
        tool_response = selected_tool.invoke(tool_call["args"])
        messages.append(ToolMessage(tool_response,
tool_call_id=tool_call["id"]))

    final_response = llm_with_tools.stream(messages)

    with st.container(height=500, border=True):
        st.write_stream(final_response)
```

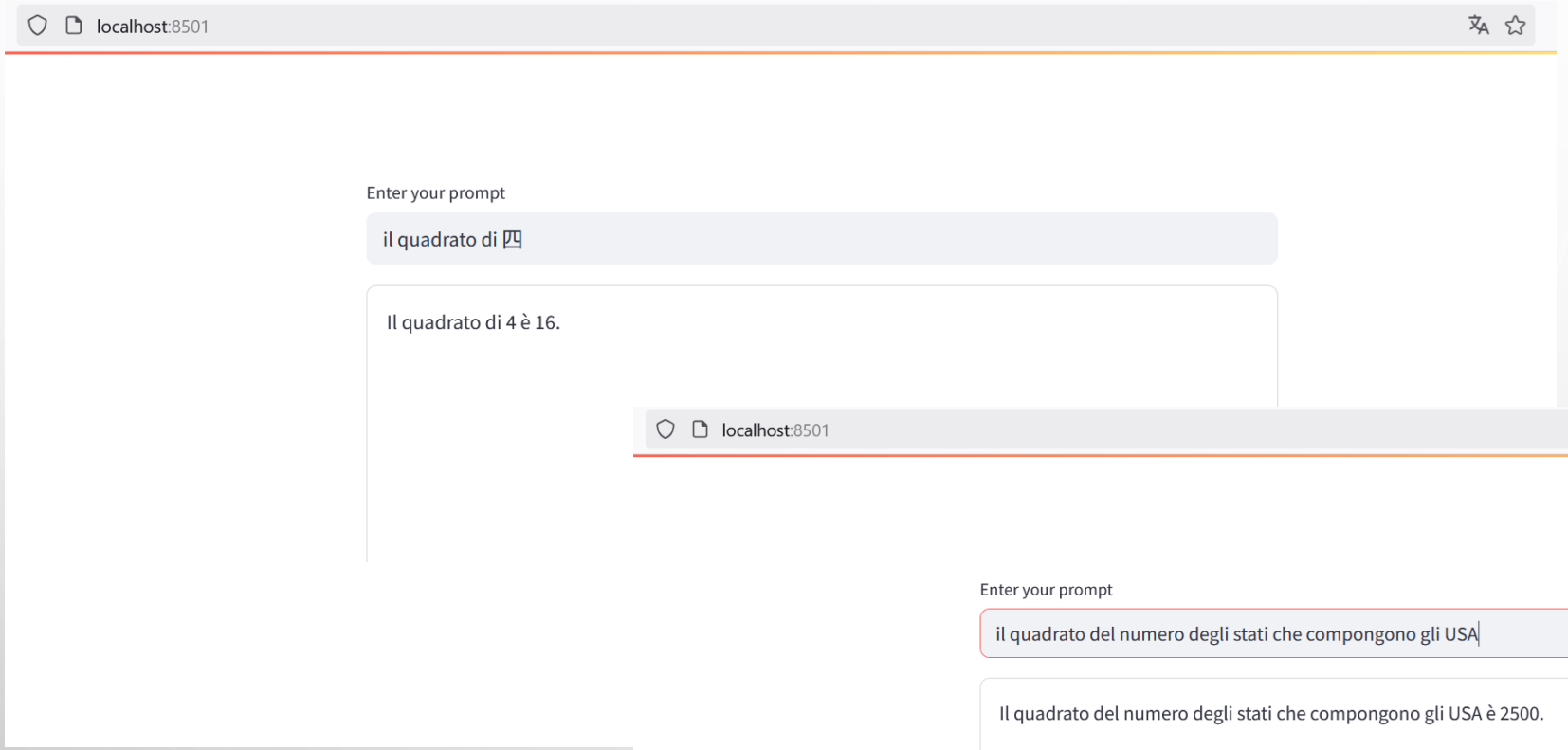
streamlit run Tool.py

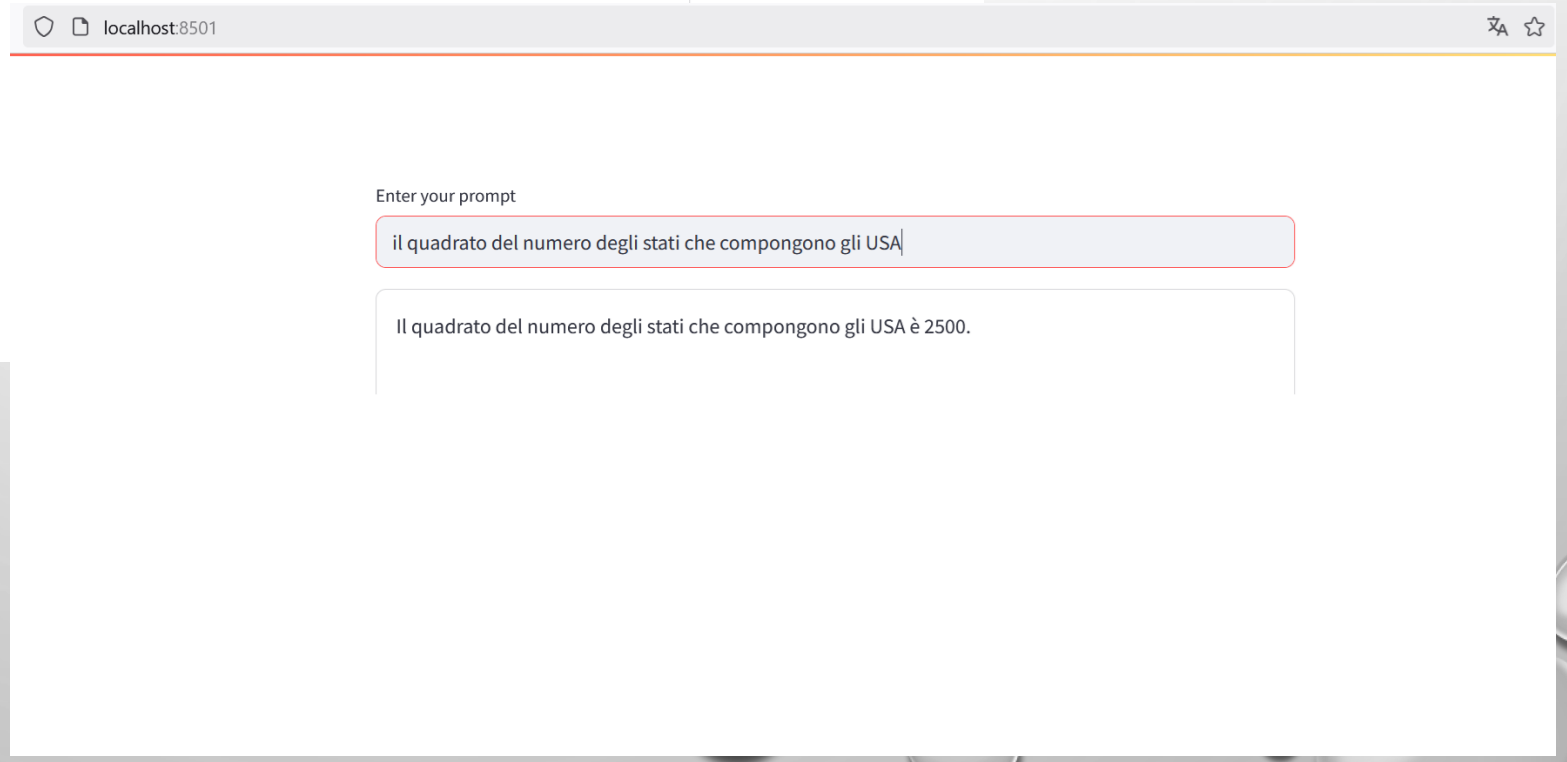
Una volta definiti i tool, vanno agganciati all'LLM (si noti l'utilizzo di ChatOllama e non Ollama per questa operazione avanzata).

Nella prima fase vengono individuate le funzioni che LLM ritiene utili generando l'oggetto ai_response

Poi, vengono invocate le funzioni in ai_response per ottenere il contesto finale da aggiungere al messaggio e darlo in pasto all'LLM per generare la risposta finale

ALCUNI ESEMPI DI UTILIZZO





LIST COMPREHENSION

La **list comprehension** è una sintassi di programmazione Python che permette di riempire una lista semplicemente inserendo il codice che la riempie all'interno delle parentesi quadre []. Ci ricorda un po' la programmazione funzionale

Ad esempio in modo 'tradizionalmente' si utilizzerebbe la seguente sintassi

```
mix = []
lista_uno = [1, 2, 3]
lista_due = [3, 1, 4]

for x in lista_uno:
    for y in lista_due:
        if x != y:
            mix.append((x, y))
```

Al contrario con la list comprehension si ha:

```
mix = [(x, y) for x in lista_uno for y in lista_due if x != y]
```

Il risultato è il seguente:

```
print(mix)
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Fonte:
https://www.programmareinpython.it/video-corso-python-intermedio/01-list-comprehensions/

RAG2 - 1

```
from langchain_ollama import ChatOllama
llm = ChatOllama(model="llama3.2:3b")

from langchain_ollama import OllamaEmbeddings
embeddings = OllamaEmbeddings(model="nomic-embed-text")

from langchain_chroma import Chroma
vector_store = Chroma(
    collection_name="example_collection",
    embedding_function=embeddings,
)

import bs4
from langchain import hub
from langchain_community.document_loaders import WebBaseLoader
from langchain_core.documents import Document
from langchain_text_splitters import RecursiveCharacterTextSplitter
from typing_extensions import List, TypedDict

loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("post-content", "post-title", "post-header")
        )
    ),
)
docs = loader.load()

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
all_splits = text_splitter.split_documents(docs)

_ = vector_store.add_documents(documents=all_splits)
```

Per poter utilizzare i tools devo usare ChatOllam

Come negli esempi precedenti il blog viene caricato e suddiviso in tanti chunk che poi vengono memorizzati nel db vettoriale

Per riferimento:
https://python.langchain.com/docs/tutorials/qa_chat_history/

RAG2 - 2

```
from langgraph.graph import MessagesState, StateGraph
graph_builder = StateGraph(MessagesState)

from langchain_core.tools import tool

@tool(response_format="content_and_artifact")
def retrieve(query: str):
    """Retrieve information related to a query."""
    retrieved_docs = vector_store.similarity_search(query, k=2)
    serialized = "\n\n".join(
        (f"Source: {doc.metadata}\n" f"Content: {doc.page_content}")
        for doc in retrieved_docs
    )
    return serialized, retrieved_docs

from langchain_core.messages import SystemMessage
from langgraph.prebuilt import ToolNode

# Step 1: Generate an AIMessage that may include a tool-call to be sent.
def query_or_respond(state: MessagesState):
    """Generate tool call for retrieval or respond."""
    llm_with_tools = llm.bind_tools([retrieve])
    response = llm_with_tools.invoke(state["messages"])
    # MessagesState appends messages to state instead of overwriting
    return {"messages": [response]}

# Step 2: Execute the retrieval.
tools = ToolNode([retrieve])
```

In questo caso, lo stato del sistema viene modellato da una sequenza di messaggi con un apposito oggetto MessageState presente nel framework che sarà lo stato all'interno del grafo dotato di stato StateGraph

Lo scopo è quello di avere uno storico della interazione tra i produttori di messaggi

Vengono distinte alcune tipologie:
HumanMessage: generate dall'utente
ToolMessage: generate dai tool
AIMessage: generate dall'LLM

Con l'indicazione # Step nel codice si intendono le funzioni che andranno a costituire il cuore dei nodi nel grafo

RAG2 - 3

```
#Step 3: Generate a response using the retrieved content.
def generate(state: MessagesState):
    """Generate answer."""
    # Get generated ToolMessages
    recent_tool_messages = []
    for message in reversed(state["messages"]):
        if message.type == "tool":
            recent_tool_messages.append(message)
        else:
            break
    tool_messages = recent_tool_messages[::-1]

    # Format into prompt
    docs_content = "\n\n".join(doc.content for doc in tool_messages)
    system_message_content = (
        "You are an assistant for question-answering tasks. "
        "Use the following pieces of retrieved context to answer "
        "the question. If you don't know the answer, say that you "
        "don't know. Use three sentences maximum and keep the "
        "answer concise."
        "\n\n"
        f"{docs_content}"
    )
    conversation_messages = [
        message
        for message in state["messages"]
        if message.type in ("human", "system")
        or (message.type == "ai" and not message.tool_calls)
    ]
    prompt = [SystemMessage(system_message_content)] + conversation_messages
    # Run
    response = llm.invoke(prompt)
    return {"messages": [response]}
```

La funzione che genera la risposta opera nel modo seguente:

Recupera il contesto selezionando dalla coda messaggi quelli provenienti dal tool (si noti la sintassi della *List Comprehension* nel codice)

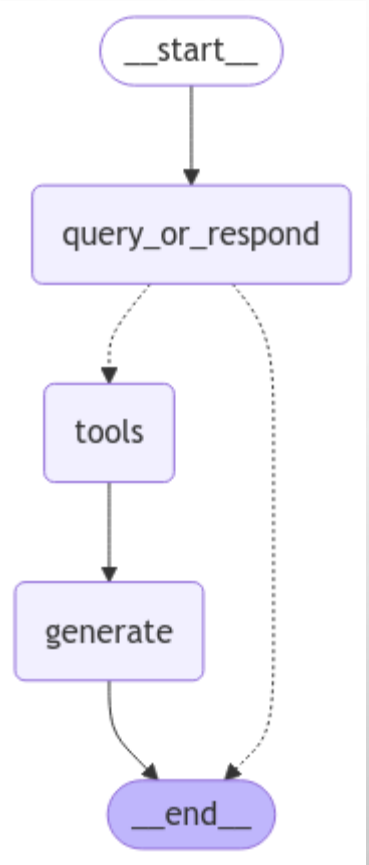
Prepara il prompt componendolo come un messaggio di sistema (come deve essere la risposta + il contesto) e i messaggi della conversazione che ci sono stati fino a quel momento

Poi genera la successiva risposta da parte dell'LLM

RAG2 - 4

```
#ultima parte

from langgraph.graph import END
from langgraph.prebuilt import tools_condition

graph_builder.add_node(query_or_respond)
graph_builder.add_node(tools)
graph_builder.add_node(generate)

graph_builder.set_entry_point("query_or_respond")
graph_builder.add_conditional_edges(
    "query_or_respond",
    tools_condition,
    {END: END, "tools": "tools"},
)
graph_builder.add_edge("tools", "generate")
graph_builder.add_edge("generate", END)

graph = graph_builder.compile()

input_message = "What is Task Decomposition?"

for step in graph.stream(
    {"messages": [{"role": "user", "content": input_message}]},
    stream_mode="values",

):
    step["messages"][-1].pretty_print()
```

Viene costruito il seguente grafo



RAG2- 5

========== HUMAN MESSAGE ==========

WHAT IS TASK DECOMPOSITION?

========== AI MESSAGE ==========

TOOL CALLS:

RETRIEVE (96A58CC6-D95D-4592-8D93-20ED6825B1CC)

ARGS:

QUERY: TASK DECOMPOSITION

========== TOOL MESSAGE ==========

NAME: RETRIEVE

SOURCE: {'SOURCE': 'HTTPS://LILIANWENG.GITHUB.IO/POSTS/2023-06-23-AGENT/'}

CHAIN OF THOUGHT (COT; WEI ET AL. 2022) HAS BECOME A STANDARD PROMPTING TECHNIQUE FOR ENHANCING MODEL PERFORMANCE ON COMPLEX TASKS. THE MODEL IS INSTRUCTED TO “THINK STEP BY STEP” TO UTILIZE MORE TEST-TIME COMPUTATION TO DECOMPOSE HARD TASKS INTO SMALLER AND SIMPLER STEPS. COT TRANSFORMS BIG TASKS INTO MULTIPLE MANAGEABLE TASKS AND SHED LIGHTS INTO AN INTERPRETATION OF THE MODEL’S THINKING PROCESS.

SOURCE: {'SOURCE': 'HTTPS://LILIANWENG.GITHUB.IO/POSTS/2023-06-23-AGENT/'}

WITH THE INPUT AND THE INFERENCE RESULTS, THE AI ASSISTANT NEEDS TO DESCRIBE THE PROCESS AND RESULTS. THE PREVIOUS STAGES CAN BE FORMED AS - USER INPUT: {{ USER INPUT }}, TASK PLANNING: {{ TASKS }}, MODEL SELECTION: {{ MODEL ASSIGNMENT }}, TASK EXECUTION: {{ PREDICTIONS }}. YOU MUST FIRST ANSWER THE USER'S REQUEST IN A STRAIGHTFORWARD MANNER. THEN DESCRIBE THE TASK PROCESS AND SHOW YOUR ANALYSIS AND MODEL INFERENCE RESULTS TO THE USER IN THE FIRST PERSON. IF INFERENCE RESULTS CONTAIN A FILE PATH, MUST TELL THE USER THE COMPLETE FILE PATH.

========== AI MESSAGE ==========

TASK DECOMPOSITION IS A TECHNIQUE USED IN PLANNING WHERE A COMPLICATED TASK IS BROKEN DOWN INTO SMALLER AND SIMPLER STEPS, ALLOWING AN AGENT TO PLAN AHEAD AND TACKLE COMPLEX TASKS MORE EFFECTIVELY. THIS IS ACHIEVED THROUGH CHAIN OF THOUGHT (COT), A STANDARD PROMPTING TECHNIQUE THAT INSTRUCTS THE MODEL TO "THINK STEP BY STEP" AND UTILIZE MORE TEST-TIME COMPUTATION.

AGENTI

- AL CONTRARIO DEGLI LLM, CHE POSSONO SOLO GENERARE TESTO IN OUTPUT, UN **AGENTE** PUÒ PRENDERE DECISIONI ED ESEGUIRE AZIONI IN BASE ALLE INFORMAZIONI OFFERTE DALL'LLM E DAI TOOL.
- UN AGENTE PUÒ PRENDERE IL CONTROLLO DEL FLUSSO ALL'INTERNO DI UN GRAFO SCEGLIENDO QUALI NODI ESEGUIRE.
- ESISTONO DIVERSE TIPOLOGIE DI AGENTI GIÀ PRONTI IN LANGGRAPH. AD ESEMPIO **REACT** È UN AGENTE CHE È IN GRADO DI DECIDERE COME PIANIFICARE LE RICHIESTE PER RISPONDERE AD UNA DOMANDA DELL'UTENTE UTILIZZANDO TRE DIVERSE RISORSE:
  - MEMORIA
  - TOOLS
  - PLANNIG

Per approfondire:
https://langchain-ai.github.io/langgraph/concepts/agentic_concepts

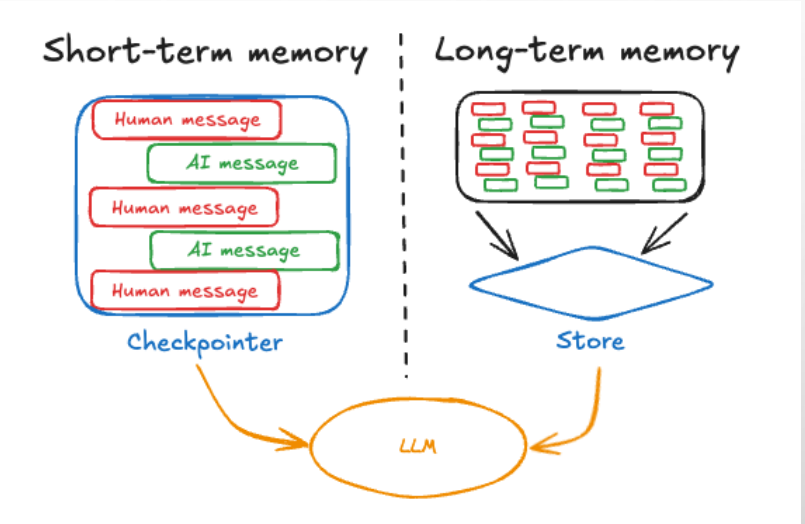
MEMORIA

La memoria permette ad un agente di trattenere e riutilizzare le informazioni durante ogni passo dell'elaborazione.

La memoria può essere a **breve termine** se interessa la sessione corrente o a **lungo termine** se interessa più sessioni nel tempo.

Langraph implementa tali concetti nel seguente modo:

- State: lo stato corrente dell'elaborazione secondo una struttura dati definita dall'utente
- CheckPointer: meccanismo che memorizza lo stato corrente durante l'elaborazione
- Store: memorizza i dati tra più sessioni

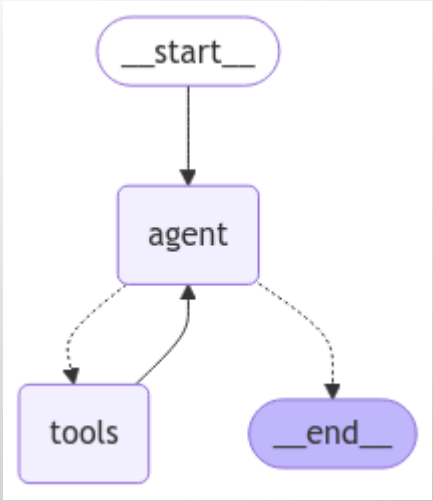


Per approfondire:
https://langchain-ai.github.io/langgraph/concepts/memory/

RAG CON AGENTE (CONTINUA DA RAG2)

```
#ultima parte di rag2

from langgraph.checkpoint.memory import MemorySaver

memory = MemorySaver()

from langgraph.prebuilt import create_react_agent

agent_executor = create_react_agent(llm, [retrieve],
checkpointer=memory)

# Specify an ID for the thread
config = {"configurable": {"thread_id": "abc123"}}

input_message = (
    "What is the standard method for Task Decomposition?\n\n"
    "Once you get the answer, look up common extensions of that
method."
)

for event in agent_executor.stream(
            {"messages": [{"role": "user", "content": input_message}]},
            stream_mode="values",
            config=config,
):
            event["messages"][-1].pretty_print()
```

Utilizzando un grafo preconfezionato (react_agent) possiamo interrogare l'LLM con domande più complesse che possono richiedere più di una interazione con la base documentale.
Sarà l'agente che 'deciderà' quante volte deve consultare la base.



TRACCIA DEL PROCESSO DI RISPOSTA

========== **Human Message** ==========

What is the standard method for Task Decomposition?

Once you get the answer, look up common extensions of that method.

========== **Ai Message** ==========

Tool Calls:
  retrieve (ab9cece2-8984-4978-9135-2cb3bab5aa09)
 Call ID: ab9cece2-8984-4978-9135-2cb3bab5aa09
  Args:
    query: standard method for Task Decomposition
Name: retrieve

========== **Tool Message** ==========

Source: {'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/'}
Content: Task decomposition can be done (1) by LLM with simple prompting like "Steps for XYZ.\n1.", "What are the subgoals for achieving XYZ?", (2) by using task-specific instructions; e.g. "Write a story outline." for writing a novel, or (3) with human inputs.
Another quite distinct approach, LLM+P (Liu et al. 2023), ….

Source: {'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/'}
Content: Component One: Planning#
A complicated task usually involves many steps. An agent needs to know what they are and plan ahead.
Task Decomposition#
Chain of thought (CoT; Wei et al. 2022) has become a standard prompting technique for enhancing model performance on complex tasks. The model is instructed to “think step by step” to utilize more test-time computation to decompose hard tasks into smaller and simpler steps. CoT transforms big tasks into multiple manageable tasks and shed lights into an interpretation of the model’s thinking process….

========== **Ai Message** ==========

The standard method for Task Decomposition is the Chain of Thought (CoT), which involves instructing the model to think step by step and decompose complex tasks into smaller, simpler steps. This technique transforms big tasks into multiple manageable tasks and sheds light on an interpretation of the model's thinking process.

Common extensions of this method include:

1. Tree of Thoughts: This extends CoT by exploring multiple reasoning possibilities at each step, generating multiple thoughts per step, and creating a tree structure.
2. LLM+P (Liu et al. 2023): This approach involves using an external classical planner to do long-horizon planning, relying on the model to translate the problem into PDDL, request a planner to generate a plan based on an existing domain-specific PDDL, and then translating the plan back into natural language.

OBBLIGHI SECONDO L'AI ACT

SE L'APPLICAZIONE RIENTRA NEL **RISCHIO LIMITATO** È SUFFICIENTE:

- AVVERTIRE L'UTENZA DEL FATTO CHE I RISULTATI SONO GENERATI DA UN IA
- ALFABETIZZARE ADEGUATAMENTE IL PERSONALE IN MERITO ALL'IA (SCADENZA IL 2 AGOSTO)

SE L'APPLICAZIONE RIENTRA **NELL'ALTO RISCHIO** (SCADENZA 2 AGOSTO 2026) SI APPLICA QUANTO RICHIESTO DAL REGOLAMENTO

- AD ESEMPIO UN SISTEMA AD ALTO RISCHIO POTREBBE ESSERE UN LLM USATO PER SELEZIONARE I CURRICULA IN AUTOMATICO (SI VEDA LA RICERCA SEMANTICA)

Per ulteriori informazioni sulle scadenze di quest'anno consiglio di vedere:
https://www.agendadigitale.eu/sicurezza/privacy/ai-act-via-ai-divieti-guida-ai-sistemi-proibiti-e-agli-obblighi-per-le-aziende/

BIBLIOGRAFIA ESSENZIALE

PIP

HTTPS://PHOENIXNAP.COM/KB/INSTALL-PIP-WINDOWS

LANGCHAIN

HTTPS://PYTHON.LANGCHAIN.COM/DOCS/INTRODUCTION/

PROMPT ENGINEERING

HTTPS://WWW.HOSTINGER.COM/TUTORIALS/AI-PROMPT-ENGINEERING?UTM_CAMPAIGN=GENERIC-TUTORIALS-DSA|NT:SE|LO:NL&UTM_MEDIUM=PPC&GAD_SOURCE=1&GCLID=EAIAIQOBCHMIQU_SOTVBIWMV8JPQBH3W8ROMEAAYASAAEGKKGPD_BWE

HTTPS://WWW.PROMPTINGGUIDE.AI/INTRODUCTION/EXAMPLES

CHATGPT + VECTOR DB

HTTPS://WWW.YOUTUBE.COM/WATCH?V=Z3UWLEYWOQA