

# LangChain

componentes e agentes

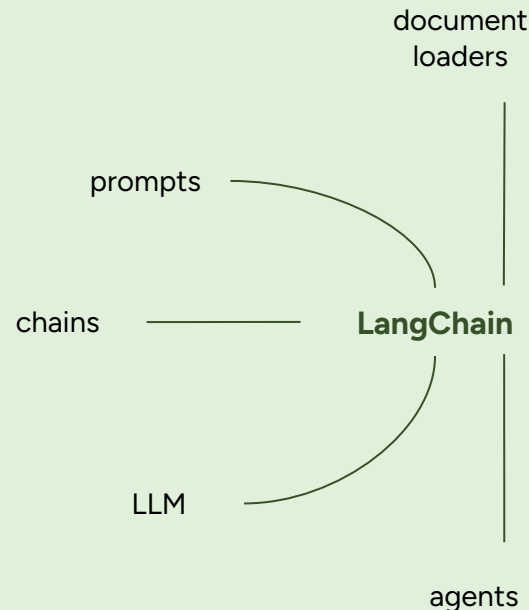
**01**



# **Sobre o LangChain**

# LangChain

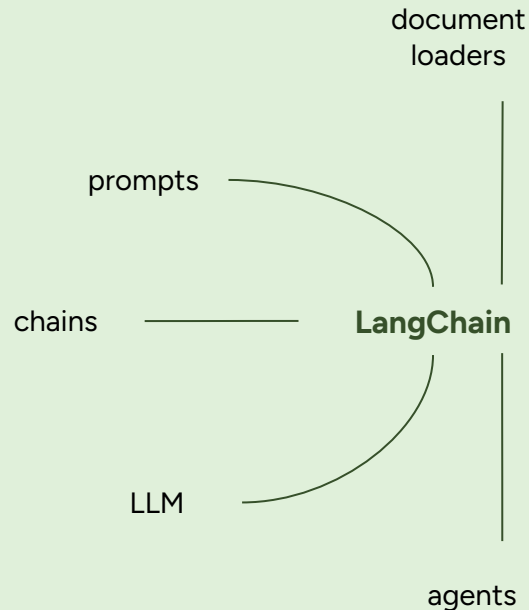
- framework de código aberto que permite criar diversas aplicações baseadas em LLMs
- criado em 2022 por Harrison Chase
  - ascensão maior em junho de 2023
  - projeto de código aberto que mais crescia no Github
  - ajudou a tornar a IA generativa mais acessível
- disponível para python e java
- possibilita a integração das aplicações LLMs com fontes externas de dados
- abordagem modular
  - usar diferentes LLMs para diferentes tarefas
- maior facilidade em comparar diferentes prompts e modelos



# LangChain

## Integrações

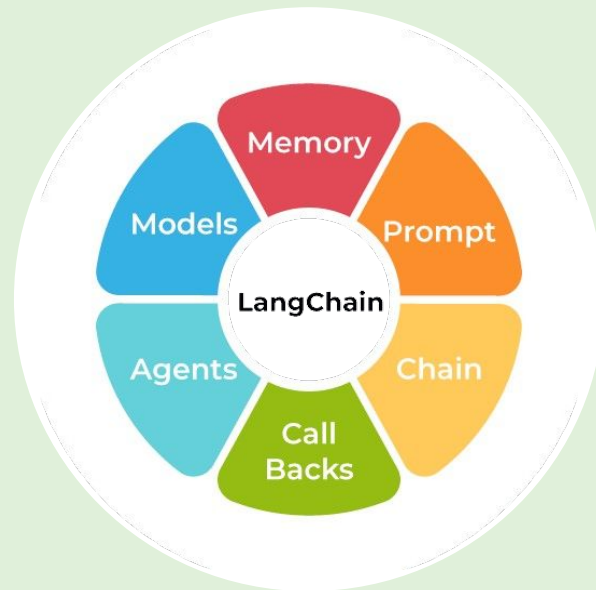
- tarefas podem precisar acessar informações contextuais específicas
  - documentação interna
  - conhecimento de domínio
- os LLMs precisam estar conectados a essas fontes externas de dados, de alguma forma
  - acesso a APIs, por exemplo
- essas integrações podem ser feitas manualmente mas o LangChain facilita o processo
  - facilita também a comparação dos resultados



# LangChain

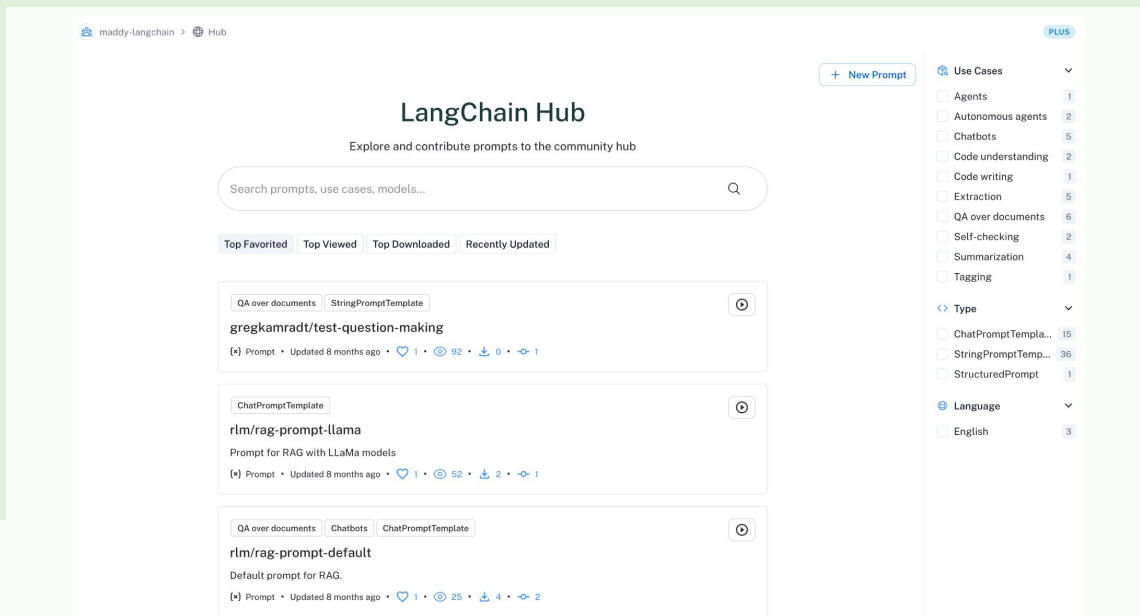
## Como funciona?

- biblioteca de abstrações
- componentes modulares (funções, classes etc)
  - servem como blocos de construção de programas de IA generativa
  - podem ser encadeados
  - minimizam a quantidade de código



# LangChain Hub

- coleção de prompts para diferentes funcionalidades
- playground para prompts
- contribuições da comunidade



**02**



# **Componentes**

# Modelos

- parte principal das aplicações
- oferece blocos para interagir com os modelos
- dois tipos de modelos
  - LLMs e ChatModels
  - se diferenciam pelos tipos de entradas e saídas



- objetos que representam uma configuração para um modelo específico
  - é possível passar parâmetros como a temperatura



# Modelos

- parte principal das aplicações
- oferece blocos para interagir com os modelos
- dois tipos de modelos
  - LLMs e ChatModels
  - se diferenciam pelos tipos de entradas e saídas



- uso dos Prompt Templates para entradas
- uso dos Output Parsers para saídas

# Modelos

- parte principal das aplicações
- oferece blocos para interagir com os modelos
- dois tipos de modelos
  - LLMs e ChatModels
  - se diferenciam pelos tipos de entradas e saídas
- uso dos Prompt Templates para entradas
- uso dos Output Parsers para saídas

## LLM

```
llm = OpenAI()  
text = "What would be a good company name for a  
company that makes colorful socks?"  
llm.invoke(text)
```

entrada  
(string)

Feetful of Fun

## Chat Models

```
chat_model = ChatOpenAI(model="gpt-3.5-turbo-0125")  
text = "What would be a good company name for a  
company that makes colorful socks?"  
  
messages = [HumanMessage(content=text)]  
chat_model.invoke(messages)
```

saída

AIMessage(content="Socks O'Color")

saída  
(string)

entrada (lista de mensagens)

# Modelos

## Tipos de mensagens

- toda mensagem possui:
  - **role**: especifica quem está enviando a mensagem (human, AI, system)
    - diferentes tipos de mensagem para diferentes roles
  - **content**: descreve o que está sendo dito na mensagem
    - string
    - uma lista de dicionários

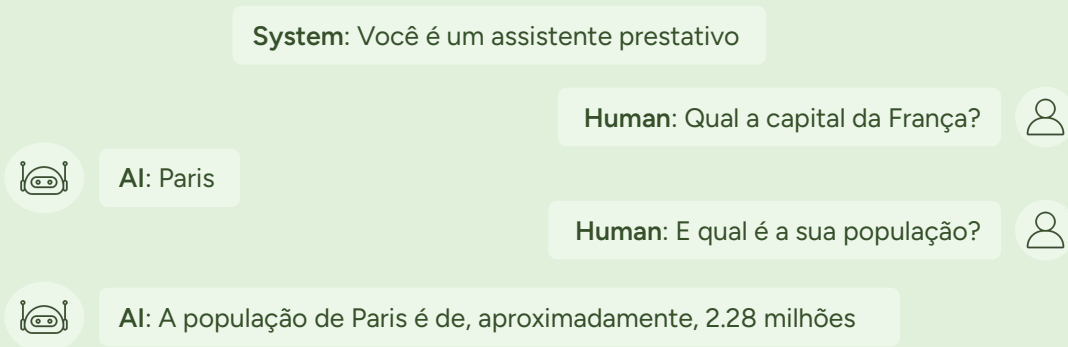
```
messages = [  
    SystemMessage(content="You're a helpful assistant"),  
    HumanMessage(content="What is the purpose of model regularization?"),  
]  
chat.invoke(messages)
```

```
AIMessage(content="The purpose of model regularization is to prevent ....")
```

# Modelos

## Tipos de mensagens

- **HumanMessage**: representa a mensagem do usuário
- **AIMessage**: representa uma mensagem do modelo
- **SystemMessage**: mensagem do sistema, diz como o modelo deve se comportar
- **FunctionMessage**: representa o resultado de uma chamada de função
- **ToolMessage**: representa o resultado de uma chamada de tool



# Modelos

## Tipos de mensagens

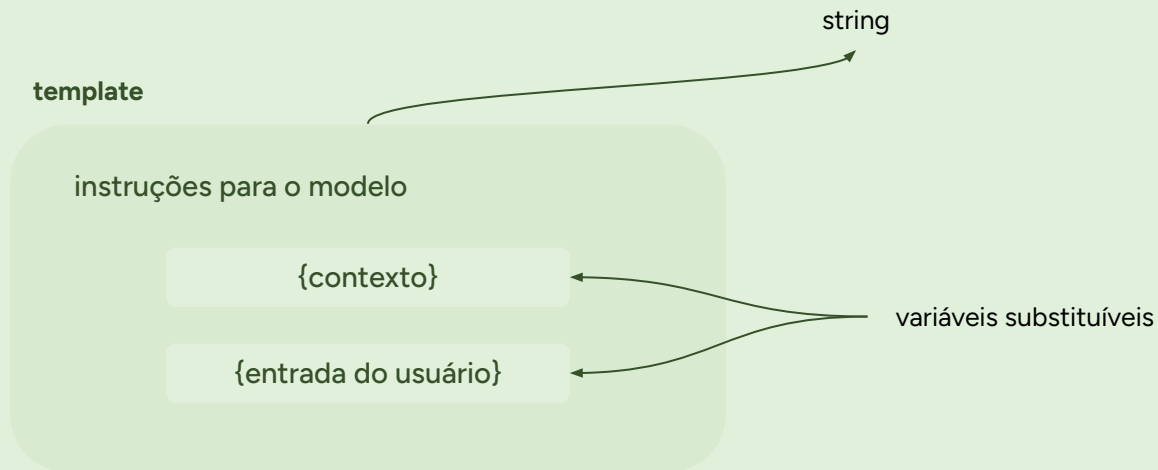
- **HumanMessage**: representa a mensagem do usuário
- **AIMessage**: representa uma mensagem do modelo
- **SystemMessage**: mensagem do sistema, diz como o modelo deve se comportar
- **FunctionMessage**: representa o resultado de uma chamada de função
- **ToolMessage**: representa o resultado de uma chamada de tool

```
messages = [  
    SystemMessage(content="You're a helpful assistant"),  
    HumanMessage(content="What is the purpose of model regularization?"),  
]  
chat.invoke(messages)
```

```
AIMessage(content="The purpose of model regularization is to prevent ....")
```

# Prompt Templates

- receitas para criar prompts para modelos de linguagem
- lógica necessária para transformar entrada ➡ prompt formatado
- entrada do usuário + contexto
- um template pode ter:
  - instruções
  - exemplos



# Prompt Templates

- receitas para criar prompts para modelos de linguagem
- lógica necessária para transformar entrada ➡ prompt formatado
- entrada do usuário + contexto
- um template pode ter:
  - instruções
  - exemplos

```
template = """Responda a seguinte pergunta do usuário  
usando o contexto fornecido.
```

```
Contexto: {contexto}  
Pergunta: {pergunta}"""
```



```
prompt_template = PromptTemplate(  
    input_variables = ["contexto", "pergunta"],  
    template = template  
)
```

```
prompt_template.format(contexto="...", pergunta="...")
```

# LCEL

- LangChain Expression Language
- criar chains
  - simplifica o processo
- integração com o LangSmith e LangServe
- pipe operator
  - saída de um componente ➡ entrada para o próximo
  - passar o que está à esquerda para a direita
- exemplo: usar o LLM para gerar um código em python
- métodos:
  - `.invoke()`: passar uma entrada e receber uma saída
  - `.batch()`: passar várias entradas para obter várias respostas
  - `.stream()`: possibilita a impressão do começo da completção antes de ela estar completa

```
chain = prompt | llm | parser
```

```
chain = retriever | prompt | llm
```

```
chain = chain1 | chain2 | chain3
```



# LCEL

- métodos:
  - `.invoke()`: passar entrada e receber saída
  - `.batch()`: passar várias entradas para obter várias respostas
  - `.stream()`: possibilita a impressão do começo da completção antes de ela estar completa

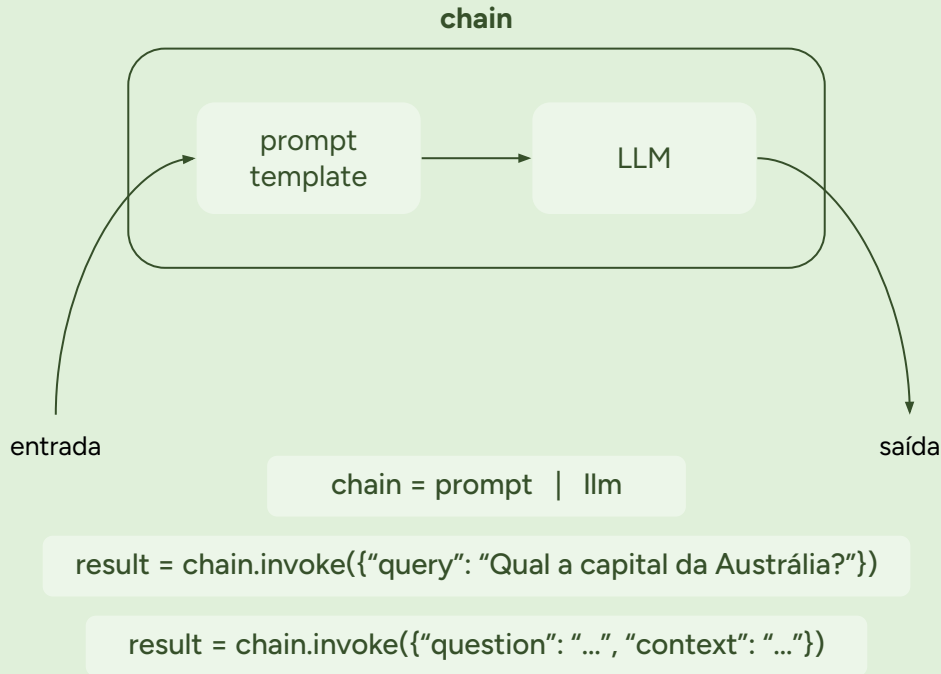
```
chain = prompt | llm
```

```
result_invoke = chain.invoke("Hello World!")  
result_batch = chain.batch(["Hello", "World", "!"])
```

```
for chunk in chain.stream("Hello World!"):  
    print(chunk, flush=True, end="")
```

# Chains

- sequência de chamadas
  - a LLMs
  - a tools
  - processamento de dados
- uso da LCEL
  - usada para construir chains
- uso de construtores
  - `create_sql_query_chain`
  - `create_retrieval_chain`
- Legacy chains
  - `LLMMath`
  - `GraphCypherQAChain`



# Exercício 1

- Crie um prompt que instrua o modelo a gerar uma sinopse para determinado filme, a partir do seu título e ano de lançamento. As variáveis *filme* e *ano de lançamento* serão substituídas depois da criação do prompt. Use o conceito de chain para encadear os componentes. Imprima a resposta do LLM.

```
prompt_template = PromptTemplate(  
    input_variables = [...],  
    template = ...  
)
```

<https://colab.research.google.com/drive/1wK2RwZWxkh9E2ljUx37o6IAyPqD1jvcS?usp=sharing>

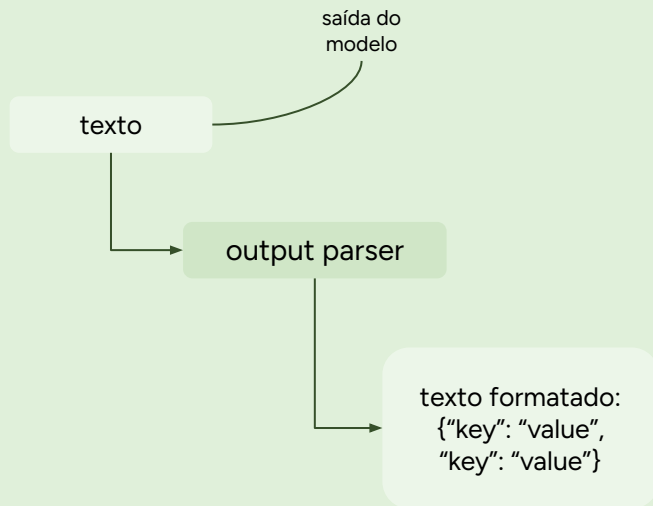
# Solução - Exercício 1

- Crie um prompt que instrua o modelo a gerar uma sinopse para determinado filme, a partir do seu título e ano de lançamento. As variáveis *filme* e *ano de lançamento* serão substituídas depois da criação do prompt. Use o conceito de chain para encadear os componentes. Imprima a resposta do LLM.

```
template = """Crie uma sinopse curta que descreva o filme a seguir, lançado no ano indicado:  
Filme: {filme}  
Ano de lançamento: {ano_lanc}""">  
prompt_template = PromptTemplate(  
    input_variables = ["filme", "ano_lanc"],  
    template = template  
)  
  
chain = prompt_template | llm  
chain.invoke({"filme": "Harry Potter e a Pedra Filosofal", "ano_lanc": "2001"})
```

# Output Parsers

- LLMs devolvem texto
- classes que ajudam a estruturar as respostas
- saída do modelo ➡ formato necessário para próxima tarefa
  - textos para informações estruturadas
    - Pydantic
    - XML
    - JSON
  - ChatMessage para string
- dois métodos
  - *get format instructions*
    - retorna string com instruções para formatação
  - *parse*
    - faz a conversão da string para o formato esperado



# Output Parsers

- LLMs devolvem texto
- classes que ajudam a estruturar as respostas
- saída do modelo ➔ formato necessário para próxima tarefa
  - textos para informações estruturadas
    - Pydantic
    - XML
    - JSON
  - ChatMessage para string
- dois métodos
  - *get format instructions*
    - retorna string com instruções para formatação
  - *parse*
    - faz a conversão da string para o formato esperado

chain = prompt | LLM | parser

# Output Parsers

```
model = ChatOpenAI(temperature=0)
```

```
class Book(BaseModel):  
    title: str = Field(description="title of the book")  
    price: float = Field(description="price of the book")
```

```
book_query = "What is George R. Martin's most famous book and its price?"  
parser = JsonOutputParser(pydantic_object=Book)  
  
prompt = PromptTemplate(  
    template="Answer the user query.\n{format_instructions}\n{query}\n",  
    input_variables=["query"],  
    partial_variables={"format_instructions": parser.get_format_instructions()},  
)  
  
chain = prompt | model | parser  
chain.invoke({"query": book_query})
```

```
{'title': 'A Game of Thrones', 'price': 12.99}
```

# Document Loaders

- carregar dados no formato de Documents
  - Document: texto + metadados
- diversos formatos de fontes
  - CSV
  - diretório
  - HTML
  - JSON
  - PDF
- muitas integrações
  - ArxiV
  - Youtube
  - Twitter

```
from langchain_community.document_loaders import UnstructuredHTMLLoader
```

```
loader = UnstructuredHTMLLoader("example_data/fake-content.html")  
data = loader.load()  
data
```

```
[Document(page_content='My First Heading\n\nMy first paragraph.', lookup_str="",  
          metadata={'source': 'example_data/fake-content.html'}, lookup_index=0)]
```



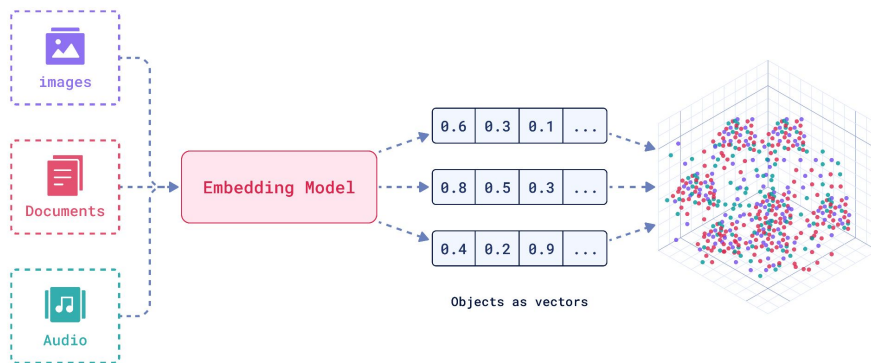
# Text Splitters

- dividir os documentos em pedaços menores (chunks)
  - cabem na janela de contexto do llm
- duas maneiras de personalizar
  - como o texto é dividido
  - como o chunk é medido
- diferentes tipos
  - **Recursive**: tentar manter pedaços de texto relacionados próximos
  - **Code**: baseia-se em caracteres específicos de linguagens de programação para dividir o texto
  - **Token**: divide o texto em tokens
  - **Character**: divide o texto usando um caractere escolhido pelo usuário

```
python_splitter = RecursiveCharacterTextSplitter.from_language(  
    language=Language.PYTHON, chunk_size=50, chunk_overlap=0  
)
```

# Modelos de Embeddings

- criam vetores que representam pedaços de textos
- possibilita a representação de textos em um espaço vetorial
- textos semelhantes devem estar próximos



# Modelos de Embeddings

- no LangChain, os modelos de embeddings possuem 2 métodos:
  - gerar embeddings a partir de documentos
    - recebe vários textos
  - gerar embeddings a partir de uma query
    - recebe um único texto

```
embeddings_model = OpenAIEmbeddings()
embeddings = embeddings_model.embed_documents(
    [
        "Hi there!",
        "Oh, hello!",
        "What's your name?",
        "My friends call me World",
        "Hello World!"
    ]
)
```

# Modelos de Embeddings

- no LangChain, os modelos de embeddings possuem 2 métodos:
  - gerar embeddings a partir de documentos
    - recebe vários textos
  - gerar embedding a partir de uma query
    - recebe um único texto

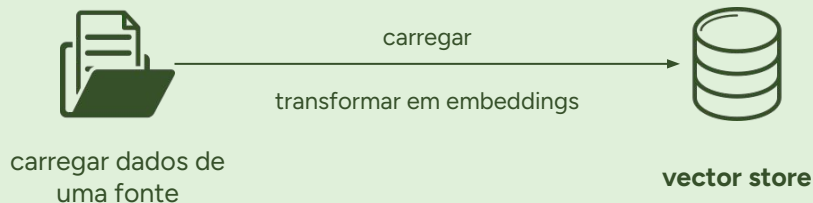
```
embedded_query = embeddings_model.embed_query("What was the name mentioned in the conversation?")
```



```
[0.0053587136790156364, -0.0004999046213924885, 0.038883671164512634, -0.003001077566295862,  
-0.00900818221271038]
```

# Vector Stores

- essencial para o processo de busca por informações relevantes
- textos são transformados em embeddings ➡ armazenados em uma vector store
- para responder uma pergunta:
  - transformar pergunta em embedding
  - buscar por vetores similares ao da pergunta
- responsável pelo armazenamento dos vetores e busca por vetores



# Vector Stores

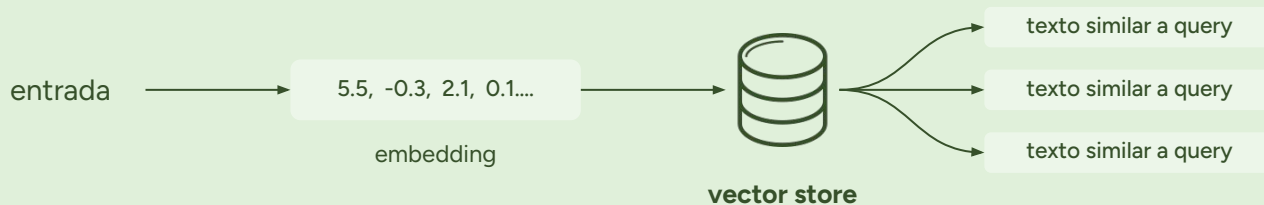
- essencial para o processo de busca por informações relevantes
- textos são transformados em embeddings ➡ armazenados em uma vector store
- para responder uma pergunta:
  - transformar pergunta em embeddings
  - buscar por vetores similares ao da pergunta
- responsável pelo armazenamento dos vetores e busca por vetores

```
raw_documents = TextLoader('.././state_of_the_union.txt').load()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
documents = text_splitter.split_documents(raw_documents)
db = Chroma.from_documents(documents, OpenAIEmbeddings())
```

```
query = "What did the president say about Ketanji Brown Jackson"
docs = db.similarity_search(query)
```

# Retrievers

- retornam documentos a partir de uma entrada
- não precisa ser capaz de armazenar documentos
- vector stores
- recebe uma string e devolve uma lista de Document's



**02**

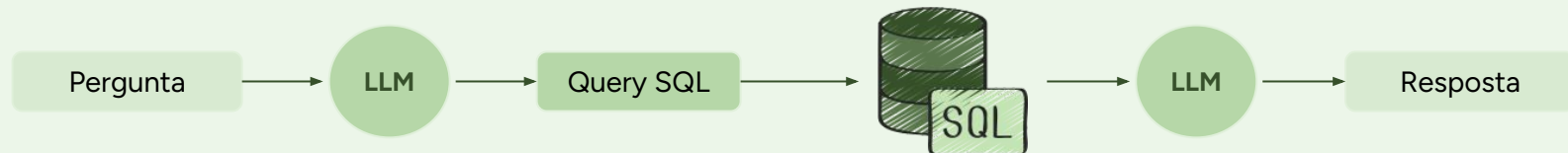


# **Aplicações**



# Q&A com SQL

- LangChain tem vários agentes e chains já prontos para lidar com SQL
- questão de segurança
  - riscos em executar consultas criadas por modelos
- duas maneiras: chain ou agente



# Q&A através de Chain

- Chain: recebe uma pergunta, transforma em uma consulta SQL, executa a consulta e usa o resultado para responder a pergunta original

1º **passo**: converter pergunta para uma consulta SQL

- a plataforma já tem um construtor para isso: **create\_sql\_query\_chain**

```
from langchain.chains import create_sql_query_chain
```

```
chain = create_sql_query_chain(llm, db)  
response = chain.invoke({"question": "How many employees are there"})  
response
```

```
chain.get_prompts()[0].pretty_print()
```



# Q&A através de Chain

- Chain: recebe uma pergunta, transforma em uma consulta SQL, executa a consulta e usa o resultado para responder a pergunta original

**2º passo:** executar a consulta SQL

- usar a tool do LangChain **QuerySQLDataBaseTool**

```
from langchain_community.tools.sql_database.tool import QuerySQLDataBaseTool

execute_query = QuerySQLDataBaseTool(db=db)
write_query = create_sql_query_chain(llm, db)
chain = write_query | execute_query
chain.invoke({"question": "How many employees are there?"})
```



# Q&A através de Chain

- Chain: recebe uma pergunta, transforma em uma consulta SQL, executa a consulta e usa o resultado para responder a pergunta original

**3º passo:** responder a pergunta

- combinar questão original e resultado da consulta para gerar resposta final
- questão original e resultado são passados para o llm novamente

```
answer = answer_prompt | llm | StrOutputParser()
chain = (
    RunnablePassthrough.assign(query=write_query).assign(
        result=itemgetter("query") | execute_query
    )
    | answer
)
chain.invoke({"question": "How many employees are there"})
```



# Q&A através de agentes

- o LangChain já tem um agente SQL que deixa as interações com banco de dados mais flexíveis

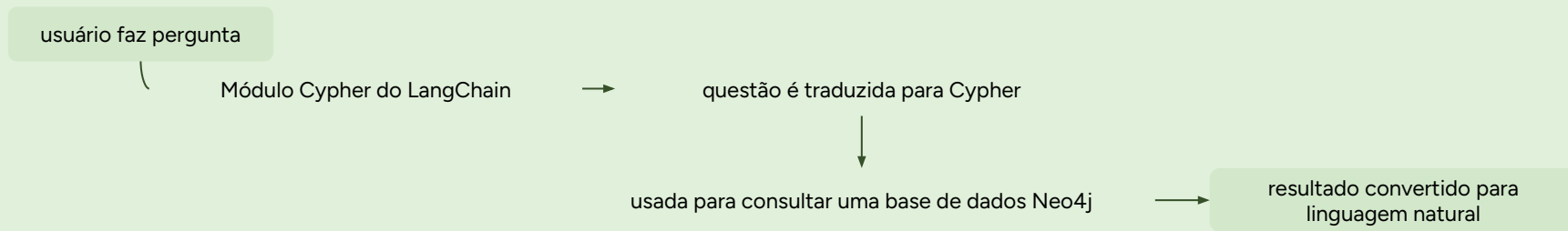
**Passo único:** criar um agente SQL usando o construtor **create\_sql\_agent**

- contém o SQLDatabaseToolkit que tem tools para:
  - criar e executar consultas
  - revisar a sintaxe da query
  - entre outras funções

```
agent_executor = create_sql_agent(llm, db=db, agent_type="openai-tools", verbose=True)
agent_executor.invoke({"input": "List the total sales per country. Which country's customers spent the most?"})
```

# Grafos

- LangChain tem vários agentes e chains que são compatíveis com linguagens de consulta de grafos
- mesma questão de segurança
- uso de chain
  - **1º passo:** converter pergunta (em linguagem natural) para uma consulta
  - **2º passo:** executar a consulta
  - **3º passo:** responder pergunta a partir do resultado da consulta



# Grafos

- LangChain tem uma chain para lidar com Neo4j: **GraphCypherQAChain**

```
from langchain.chains import GraphCypherQAChain
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)
chain = GraphCypherQAChain.from_llm(graph=graph, llm=llm, verbose=True)
response = chain.invoke({"query": "What was the cast of the Casino?"})
```

pergunta em linguagem natural

'What was the cast of the Casino?'

resposta em linguagem natural

'The cast of Casino included Joe Pesci, Robert de Niro, Sharon Stone and James Woods.'

# Grafos

## Exemplo

- conexão com um grafo do Neo4j através do LangChain



```
from langchain_community.graphs import Neo4jGraph
graph = Neo4jGraph(
    url="bolt://44.193.17.131:7687",
    username="neo4j",
    password="recruit-envelope-runoffs"
)
```



# Grafos

## Exemplo

- criação de um template para prompt que vai guiar o LLM (dentro da chain)

```
CYPHER_GENERATION_TEMPLATE = """
```

```
    You are an expert Neo4j Developer translating user questions into Cypher to answer  
    questions about movies and provide recommendations.  
    Convert the user's question based on the schema.
```

```
    Schema: {schema}
```

```
    Question: {question}
```

```
    """
```



# Grafos

## Exemplo

- criação do template através do PromptTemplate
- criação da chain para consulta de grafos

```
cypher_generation_prompt = PromptTemplate(  
    template=CYPHER_GENERATION_TEMPLATE,  
    input_variables=["schema", "question"],  
)
```



```
cypher_chain = GraphCypherQAChain.from_llm(  
    llm,  
    graph=graph,  
    cypher_prompt=cypher_generation_prompt,  
    verbose=True  
)
```




# Grafos

## Exemplo

```
cypher_generation_prompt = PromptTemplate(  
    template=CYPHER_GENERATION_TEMPLATE,  
    input_variables=["schema", "question"],  
)
```

```
cypher_chain = GraphCypherQAChain.from_llm(  
    llm=llm,  
    graph=graph,  
    cypher_prompt=cypher_generation_prompt,  
    verbose=True  
)
```



```
cypher_chain.invoke({"query": "Who plays the character 'Woody' in Toy Story?"})
```

</>

# Grafos

## Exemplo

```
cypher_generation_prompt = PromptTemplate(
    template=CYPHER_GENERATION_TEMPLATE,
    input_variables=["schema", "question"],
)
```

```
cypher_chain = GraphCypherQAChain.from_llm(
    llm=llm,
    graph=graph,
    cypher_prompt=cypher_generation_prompt,
    verbose=True
)
```

```
cypher_chain.invoke({"query": "How many movies is 'Tom Hanks' in?"})
```

# Construindo knowledge graphs

- em alto nível, os passos são:
  - usar modelo para extrair informações estruturadas do texto
  - armazenar dentro de um graph database
- uso do **LLMGraphTransformer** (langchain experimental)
  - converte documentos de texto em documentos estruturados de grafos
  - usa um LLM para analisar e categorizar entidades e seus relacionamentos
  - escolha do LLM é importante

```
from langchain_experimental.graph_transformers import LLMGraphTransformer  
from langchain_openai import ChatOpenAI  
  
llm = ChatOpenAI(temperature=0, model_name="gpt-4-turbo")  
llm_transformer = LLMGraphTransformer(llm=llm)
```

# Construindo knowledge graphs

```
llm_transformer = LLMGraphTransformer(llm=llm)  
graph_documents = llm_transformer.convert_to_graph_documents(documents)
```

- é possível definir tipos de nós específicos e relacionamentos para a extração

```
llm_transformer_filtered = LLMGraphTransformer(  
    llm=llm,  
    allowed_nodes=["Person", "Country", "Organization"],  
    allowed_relationships=["NATIONALITY", "LOCATED_IN", "WORKED_AT", "SPOUSE"],  
)
```

# Construindo knowledge graphs

- o parâmetro **node\_properties** habilita a extração de propriedades dos nós pelo LLM
  - criação de um grafo mais detalhado
  - se o parâmetro receber uma lista de strings, o LLM pega do texto apenas as propriedades especificadas

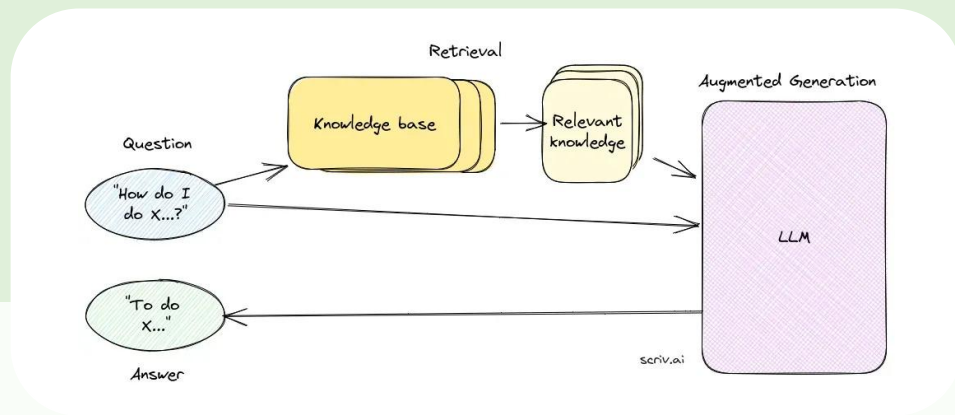
```
llm_transformer_props = LLMGraphTransformer(  
    llm=llm,  
    allowed_nodes=["Person", "Country", "Organization"],  
    allowed_relationships=["NATIONALITY", "LOCATED_IN", "WORKED_AT", "SPOUSE"],  
    node_properties=["born_year"],  
)  
  
graph_documents_props = llm_transformer_props.convert_to_graph_documents(documents)
```

- armazenar em um graph database

```
graph = Neo4jGraph()  
graph.add_graph_documents(graph_documents_props)
```

# Q&A com RAG

- aplicações que são capazes de responder perguntas sobre uma fonte de informações específica
  - usam RAG
- RAG é uma técnica para ampliar o conhecimento do LLM, através do uso de dados adicionais
- ajuda a atenuar as limitações do LLM
  - alucinações
  - período de treinamento
- maneira de fornecer os dados necessários para que o LLM responda perguntas sobre um domínio
- o LangChain tem vários componentes que auxiliam a criação de aplicações com RAG
- duas fases: indexação; recuperação e geração





# Q&A com RAG

## Indexação

- carregar os documentos ➡ uso dos **DocumentLoaders**
- dividir os documentos em chunks ➡ uso dos **TextSplitters**
- armazenar os chunks ➡ uso de **VectorStore** e um modelo de **embeddings**

## Recuperação e geração

- recuperação: baseando-se em uma entrada do usuário, pedaços de documentos relevantes são buscados ➡ uso do **Retriever**
- geração: um **LLM** gera uma resposta a partir de um prompt que inclui a entrada e os pedaços de documentos obtidos

# Q&A com RAG

## Carregar documentos

- uso dos **DocumentLoaders** ➡ objetos que carregam dados de uma fonte e retornam uma lista de documentos
- um documento é um objeto com os atributos *page\_content* e *metadata*
- no exemplo, será utilizado o WebBaseLoader

```
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()
```



# Q&A com RAG

## Dividir os documentos

- documentos são muito grandes para caberem na janela de contexto dos LLMs
- dividir o documento em chunks para gerar embeddings e para o futuro armazenamento
- uso do ***RecursiveCharacterTextSplitter*** ➡ divide o documento recursivamente baseando-se em separadores comuns (como quebra de linha)



```
from langchain_text_splitters import RecursiveCharacterTextSplitter

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

# Q&A com RAG

## Armazenar

- gerar embeddings para cada pedaço do documento
- será utilizado para a busca por similaridade (similaridade com a query do usuário)
- uso do **Chroma** e **HuggingFaceEmbeddings**

```
embeddings = HuggingFaceEmbeddings()  
vectorstore = Chroma.from_documents(documents=splits,  
embedding=embeddings)
```

# Q&A com RAG

## Recuperar

- pergunta do usuário ➡ buscar documentos relevantes ➡ passar a pergunta e os documentos para o LLM ➡ retornar resposta
- o LangChain oferece uma interface **Retriever** que retorna documentos relevantes quando recebe uma entrada de texto
- a maneira mais comum consiste em usar **VectorStoreRetriever**
  - transformação de uma VectorStore em Retriever
  - usa busca por similaridade



```
retriever = vectorstore.as_retriever()
```

# Q&A com RAG

## Gerar resposta

- juntar tudo em uma chain que recebe a pergunta, busca os documentos, compõe um prompt, passa para o modelo e converte a saída



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

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

```
rag_chain.invoke("What is Task Decomposition?")
```

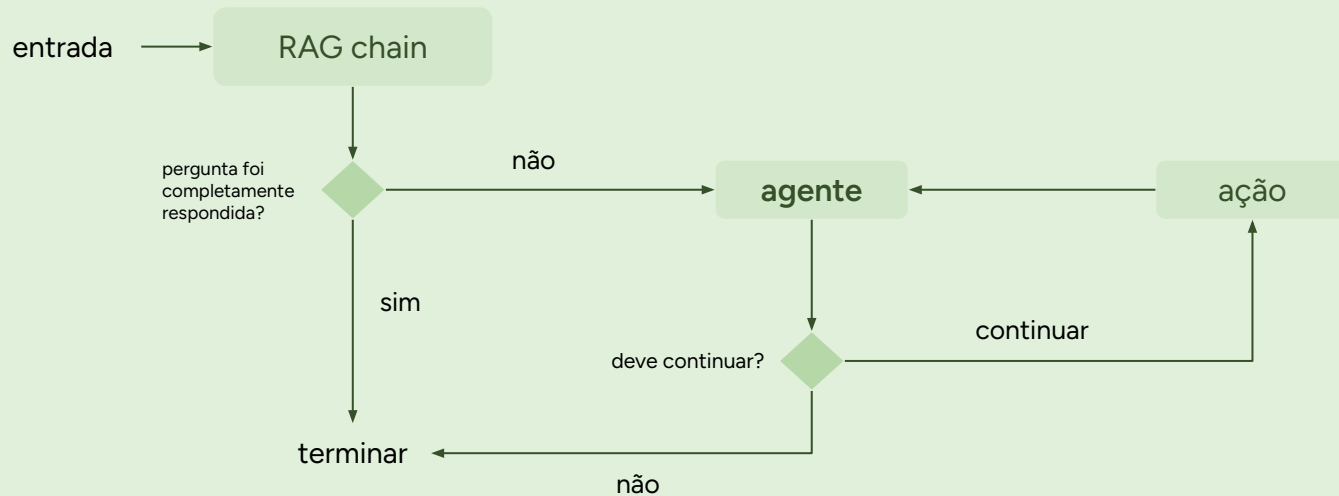
# Agentic RAG

- abordagem baseada em agentes para responder perguntas usando documentos como contexto
- possível resolução de perguntas mais complexas (passo-a-passo)
- agentes podem “raciocinar” e tomar decisões a partir dos dados
  - fazer síntese dos documentos relevantes
  - identificar o tema do documento
  - recuperar documentos com o mesmo tema



# Agentic RAG

## Possível abordagem





**03**



**Agentes**

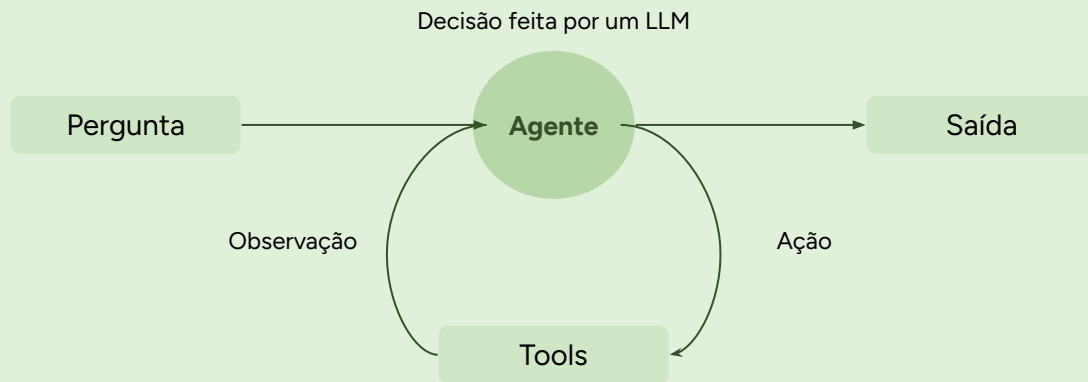
# Agente

- chain que usa LLM para escolher uma sequência de ações
- o LLM é usado para tomar decisões
  - decide qual deve ser a entrada de uma tool, por exemplo
- define em que ordem essas ações devem ser executadas
- geralmente, é também composto por um prompt e um output parser



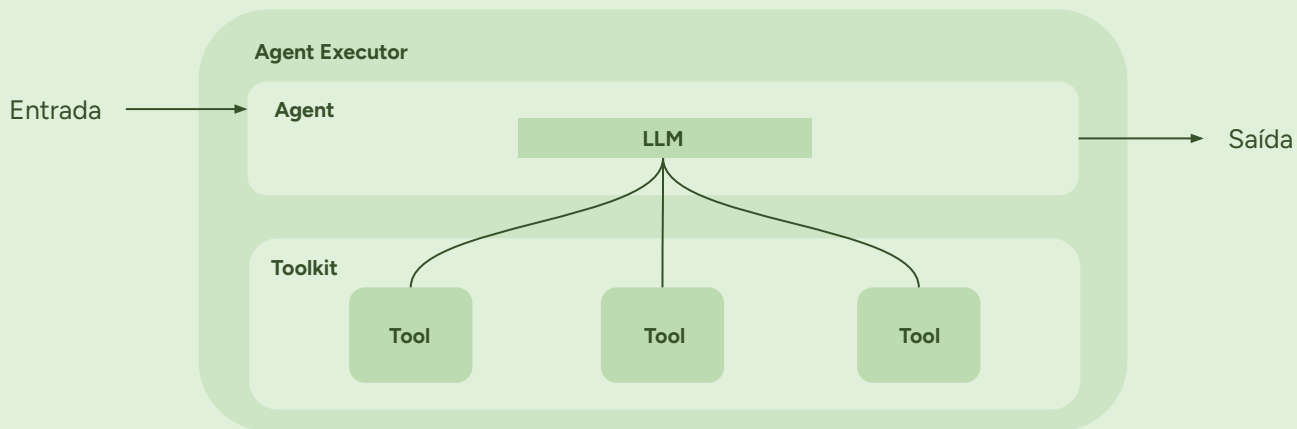
# Agente

- loop do agente
  - escolher uma tool para utilizar
  - observar o resultado devolvido pela tool
  - repetir até que uma condição de parada seja atingida



# Agente

- AgentExecutor: ambiente de execução
  - implementa o loop do agente
  - executa as ações que o agente escolheu
  - passa o resultado da ação de volta para o agente



# Agente

- AgentExecutor: ambiente de execução
  - implementa o loop do agente
  - executa as ações que o agente escolheu
  - passa o resultado da ação de volta para o agente

quantos ciclos de ação o  
agente pode ter

```
agent_executor = AgentExecutor(  
    agent=agent,  
    tools=tools,  
    memory=memory,  
    max_interations=3,  
    verbose=True,  
    handle_parse_errors=True  
)
```

# Criar agente

- definir as tools que serão utilizadas

```
tools = [search, retriever_tool]
```

- escolher um LLM para guiar o agente

```
llm = ChatOpenAI(model="gpt-3.5-turbo-0125", temperature=0)
```

- escolher o prompt para guiar o agente

```
prompt = hub.pull("hwchase17/openai-functions-agent")
```

- inicializar o agente, com o prompt e as tools

```
from langchain.agents import create_tool_calling_agent  
agent = create_tool_calling_agent(llm, tools, prompt)
```

# Criar agente

- combinar o agente com as tools dentro do AgentExecutor

```
from langchain.agents import AgentExecutor
```

```
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

- executar o agente

```
agent_executor.invoke({"input": "what is the weather in sf?"})
```

- Obs: também é possível adicionar memória para que o agente lembre de interações passadas

**04**

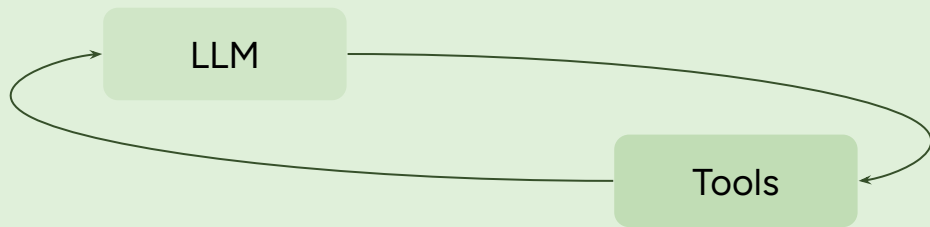


**Tools**



# Tools

- abstrações em volta de funções, que facilitam a interação com o LLM
- uma “tool” tem 2 componentes:
  - parâmetros necessários para chamar a ferramenta
  - a função
- Toolkits
  - conjunto de tools para realizar alguma tarefa



# Tools

## Toolkits

- acessados através das integrações disponibilizadas pelo LangChain
- exemplo: Toolkit que permite acesso a API da biblioteca de imagens e vídeos da NASA



```
from langchain_community.agent_toolkits.nasa.toolkit import NasaToolkit
from langchain_community.utilities.nasa import NasaAPIWrapper
```

```
nasa = NasaAPIWrapper()
toolkit = NasaToolkit.from_nasa_api_wrapper(nasa)
```

```
prompt = hub.pull("hwchase17/react")
agent_nasa = create_react_agent(llm, toolkit.get_tools(), prompt)
agent_executor_nasa = AgentExecutor(agent=agent_nasa, tools=toolkit.get_tools(), verbose=True)
```

# Tools

## Integrações

- algumas das integrações disponibilizadas pelo LangChain:
  - Github
  - Office365
  - Slack
  - Gmail

Tools disponíveis no **toolkit Gmail**:

- GmailCreateDraft
- GmailSendMessage
- GmailSearch
- GmailGetMessage
- GmailGetThread

# Criar tool personalizada

- componentes de uma tool:
  - name (obrigatório e único)
  - description (obrigatória)
    - usada pelo agente para determinar qual tool usar
  - args\_schema (opcional)
    - pode ser usado para passar informações adicionais
- existem várias maneiras de criar uma tool
  - através de um decorator
  - subclasse de BaseTool
  - StructuredTool

```
math_tool = Tool(  
    name= "Calculator"  
    description="Useful for..."  
    func=llm_math.run  
)
```

# @tool decorator

- maneira mais simples
- converte automaticamente a função em uma tool
- o decorator usa o nome da função como nome da tool (padrão)
- usa a docstring como descrição da tool

```
@tool
def multiply(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b
```



```
print(multiply.name)
print(multiply.description)
print(multiply.args)
```

# subclasse de BaseTool

- mais trabalhosa mas garante maior controle na definição da tool
- uso da função \_run

```
class CustomCalculatorTool(BaseTool):
    name="Calculator"
    description = "useful for when you need to answer questions about math"
    args_schema: Type[BaseModel] = CalculatorInput

    def _run(
        self, a: int, b: int, run_manager: Optional[CallbackManagerForToolRun] = None
    ) -> str:
        """Use the tool."""
        return a * b

    async def _arun....
```

# StructuredTool

- misto das últimas duas opções
- mais conveniente que usar a classe BaseTool
- mais funcionalidades do que o decorator

```
def search_function(query: str):  
    return "LangChain"  
  
search = StructuredTool.from_function(  
    func=search_function,  
    name="Search",  
    description="useful for when you need to answer questions about current events."  
)
```

## Exercício 2

- Crie uma tool que receba um texto e faça sua tradução para português. Use o decorator para a conversão da função em tool. Confira se os atributos da tool (name, description, args) estão corretos. Chame a tool utilizando o método “run” e passando o texto que deve ser traduzido.

valores esperados dos atributos:

```
traducao
Utilizada para a tradução de textos
{'text': {'title': 'Text', 'type': 'string'}}
```



## Solução - Exercício 2

- Crie uma tool que receba um texto e faça sua tradução para português. Use o decorator para a conversão da função em tool. Confira se os atributos da tool (name, description, args) estão corretos. Chame a tool utilizando o método "run" e passando o texto que deve ser traduzido.

```
@tool
def traducaao(text: str) -> str:
    """Utilizada para a tradução de textos"""
    prompt = f"""Traduza o seguinte texto para português:
    Texto: {text}
    Tradução: """

    answer = llm.invoke(prompt)
    return answer

print(translation.name)
print(translation.description)
print(translation.args)
traducaao.run("LangChain is a framework used for generative AI")
```

**05**

**Outras  
possibilidades**



# Agente

## Adicionar memória ao agente

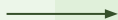
- para aplicações com aspecto conversacional
- uma das maneiras possíveis:
  - alterar o prompt do agente para que tenha uma variável para inserção do histórico: **{chat\_history}**
  - uso de **ConversationBufferMemory** ➔ armazena as mensagens, que são extraídas para uma variável
  - passar a variável como o parâmetro **memory** do AgentExecutor

no prompt:

"...{chat\_history}

Question: {input}

Thought:{agent\_scratchpad}..."



```
memory = ConversationBufferMemory(memory_key="chat_history")
agent_memory = create_react_agent(llm, tools, prompt)
agent_executor_memory = AgentExecutor(agent=agent_memory,
tools=tools, memory=memory, verbose=True)
```



# Agente

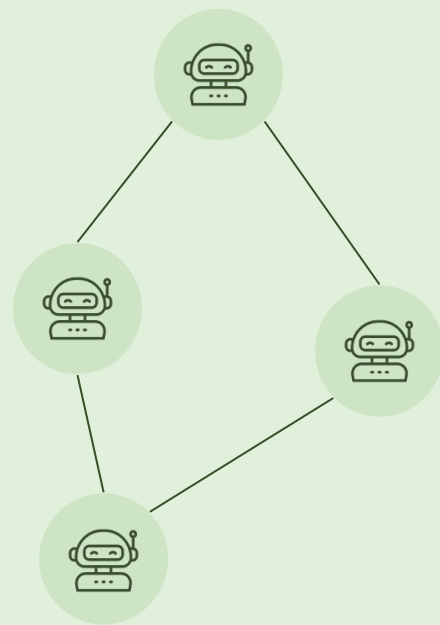
## Adicionar memória ao agente

- outros tipos de memória:
  - **ConversationBufferWindowMemory** ➡ mantém uma lista de interações ao longo do tempo, mas usa apenas as K últimas
  - **ConversationKGMemory** ➡ usa um grafo de conhecimento para recriar memória
  - **ConversationSummaryMemory** ➡ gera um sumário da conversa, e é mais útil para conversas mais longas
  - **ConversationSummaryBufferMemory** ➡ gera um sumário da conversa e usa um buffer com as últimas interações
  - **ConversationTokenBufferMemory** ➡ mantém as interações mais recentes, porém usa a quantidade de tokens para determinar quando eliminar interações mais antigas

```
memory = ConversationBufferWindowMemory(k=1)
memory = ConversationKGMemory(llm=llm)
memory = ConversationSummaryMemory(llm=OpenAI(temperature=0))
memory = ConversationSummaryBufferMemory(llm=llm, max_token_limit=10)
memory = ConversationTokenBufferMemory(llm=llm, max_token_limit=10)
```

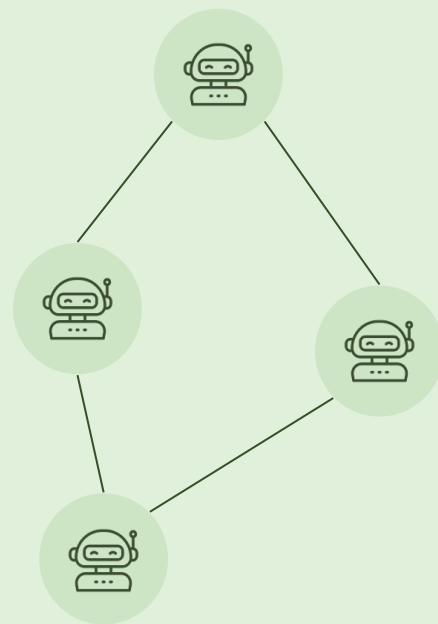
# LangGraph

- biblioteca do LangChain
- módulo construído para possibilitar a criação de chains cíclicas
- útil para criação de agentes mais controláveis
  - sempre chamar certa tool primeiro, por exemplo
  - definir como as tools devem ser chamadas
  - diferentes prompts para o agente
- fluxos mais controlados são chamados “state machines”
- LangGraph é um jeito de criar essas “state machines” na forma de grafos
- um grafo pode ter:
  - vários agentes
  - várias partes de um agente
  - várias chains



# LangGraph

- conceitos fundamentais:
  - **grafo com estado:** grafo mantém um estado que é transmitido e atualizado à medida que o processamento avança
  - **nós:** cada nó representa uma função ou um passo do processamento
    - definir nós para executar tarefas como processamento de entradas, tomada de decisões ou interações com APIs
  - **arestas:** conectam os nós do grafo, definindo o fluxo do processamento
    - podem ser arestas condicionais: permite determinar, dinamicamente, o próximo nó a ser executado baseando-se no estado atual do grafo



# LangGraph

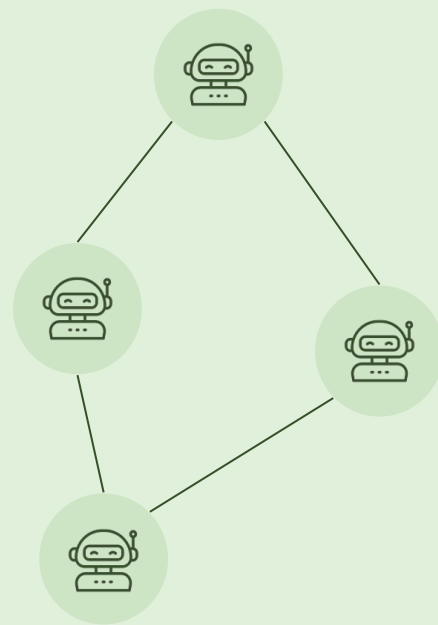
- na prática:
  - **StateGraph** ➔ classe que representa o grafo
    - para instanciá-la, é necessário passar o **State** como parâmetro, que é atualizado pelos nós do grafo

```
graph = StateGraph(State)
```
  - **add\_node** ➔ para adicionar um nó ao grafo

```
graph.add_node("model", model)
```
  - **add\_edge** ➔ para adicionar uma aresta no grafo

```
graph.add_edge("tools", "model")
```
  - **compile** ➔ compilar o grafo em um executável

```
app = graph.compile()
```



# Plan-and-Execute Agents

- implementados através do LangGraph
- quando comparados com os agentes mais simples, podem:
  - executar um fluxo de trabalho de múltiplas etapas mais rapidamente
  - oferecer redução de custos
  - oferecer melhor performance por terem que “pensar” sobre os passos a serem tomados para realizar uma tarefa
- os agentes típicos (baseados em **ReAct**):
  - propõe uma ação ➡ o LLM gera um texto para responder diretamente o usuário ou para passar para uma função
  - executa a ação ➡ o código invoca outros programas
  - observa ➡ reage a resposta da tool chamada, ao chamar outra função ou responder o usuário
  - “um passo de cada vez”



# Plan-and-Execute Agents

Ainda sobre os agentes baseados em ReAct:

**Pensamento:** I should call Search() to see the current score of the game.

**Ação:** Search("What is the current score of game X?")

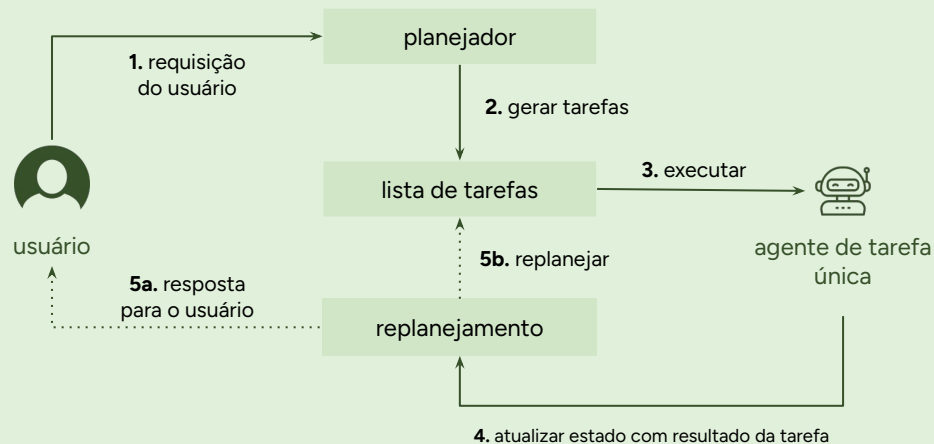
**Observação:** The current score is 24-21

... (repeat N times)

- tem alguns contras:
  - precisa do LLM para toda chamada de tool
  - o LLM planeja apenas um 1 sub-problema por vez
    - pode levar a caminhos menos eficientes

# Plan-and-Execute Agents

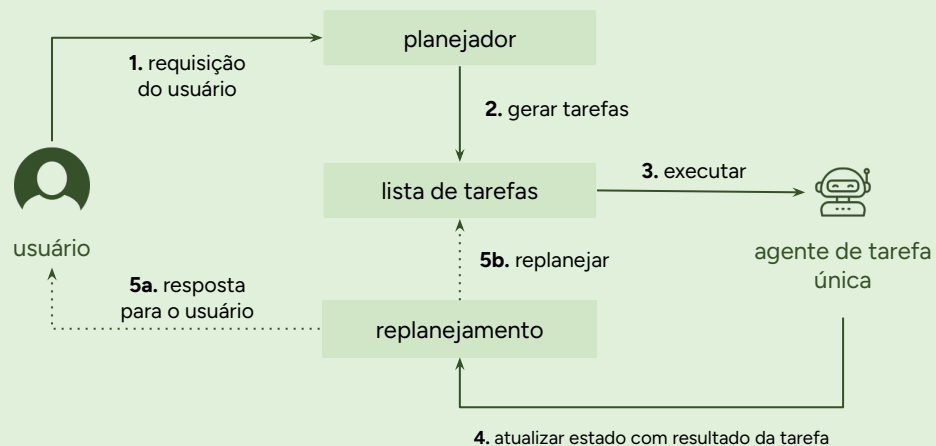
- consiste em dois componentes:
  - um **planejador** que pede para o LLM gerar um plano com múltiplos passos para realizar uma tarefa grande
  - um **executor** que recebe a entrada do usuário e uma etapa do plano, e chama uma ou mais tools para completar essa tarefa
- quando a execução termina, o agente é chamado novamente com um prompt de **replanejamento**, deixando ele decidir se deve terminar com uma resposta final ou gerar um próximo plano (caso o primeiro não tenha sido 100% efetivo)
- a estrutura desse agente evita que seja necessário chamar o LLM planejador para qualquer invocação de tool



# Plan-and-Execute Agents

## Vantagens

- planejamento explícito e de longo prazo
- capacidade de usar modelos menores para a execução de um passo, e usar maiores para a fase de planejamento

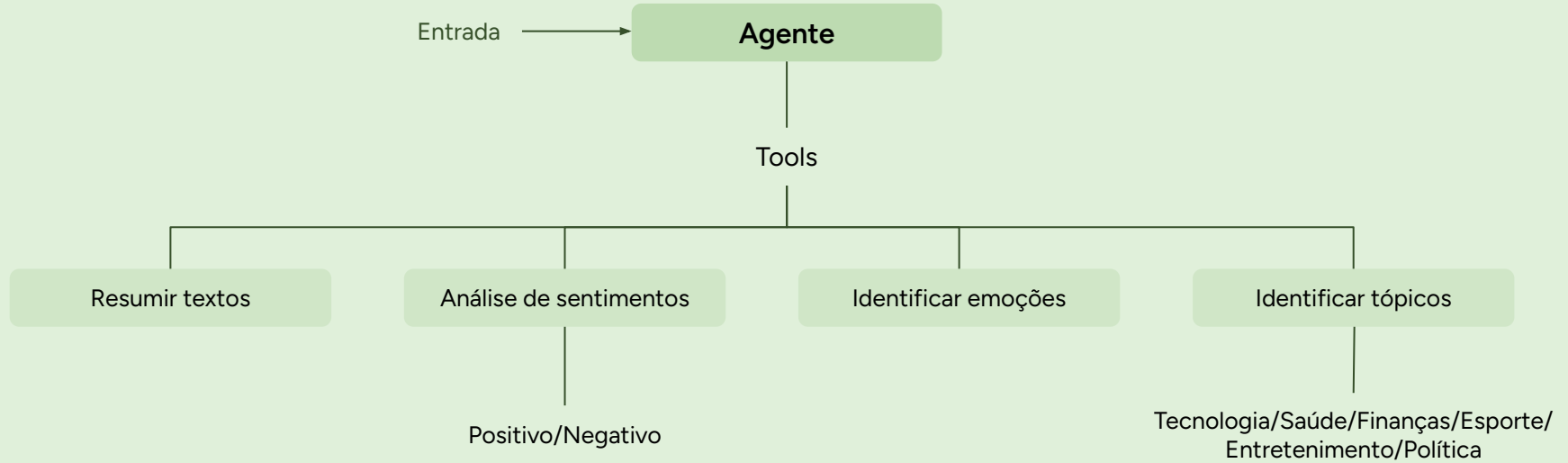


**06**



**Exemplo**

# Exemplo de agente



# criar agente



```
from langchain.agents import create_react_agent, AgentExecutor

tools = [analyze_sentiment, summarize_text, classify_emotion, identify_topic]

prompt = hub.pull("hwchase17/react")
agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

# exemplos de funcionamento

```
result = agent_executor.invoke({"input": "What is the sentiment of the phrase: 'That was a fun movie!'"})
```

Análise de sentimentos

```
result = agent_executor.invoke({"input": "Make the follow text shorter: 'Interstellar is an epic journey that transcends the bounds of space and time. Directed by Christopher Nolan, the film immerses viewers in a dystopian future where Earth faces a devastating environmental crisis. Amidst this turmoil, Cooper, a former pilot turned farmer, is recruited to lead an interstellar expedition in search of a new home for humanity.' "})
```

Resumir textos

```
result = agent_executor.invoke({"input": "Classify the emotion expressed by the phrase: 'I did not have fun watching the movie!'"})
```

Identificar emoções

```
result = agent_executor.invoke({"input": "What are the topics on the following text: 'Streaming services offer a range of digital content like movies, TV shows, music, and podcasts for on-demand consumption, typically through subscription or pay-per-use models.'"})
```

Identificar tópicos

**07**

# **Componentes Operacionais**





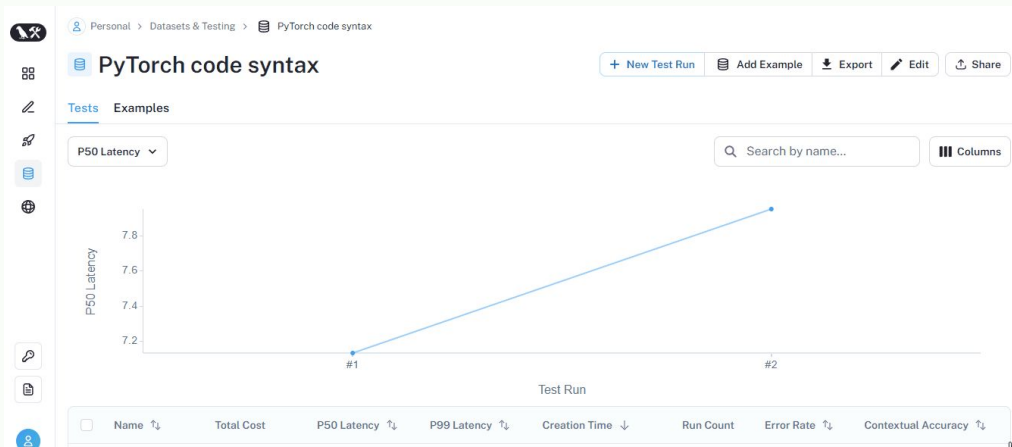
# Componentes operacionais

- o LangChain é um framework completo, já que permite:
  - construir, implementar e monitorar as aplicações criadas
- oferece vários componentes para que você escolha os mais adequados para seu projeto



# LangSmith

- ajuda a rastrear e avaliar aplicações LLM
  - agentes inteligentes
- facilita testes, debug e melhorias nas aplicações LLM
- também pode ser útil para:
  - criar e gerenciar datasets
  - capturar análises de produção para melhorias



# LangSmith

## Conexão com o LangSmith

habilitar rastreamento

```
os.environ["LANGCHAIN_TRACING_V2"] = "true"  
os.environ["LANGCHAIN_PROJECT"] = f"LangChain_Presentation"  
os.environ["LANGCHAIN_ENDPOINT"] = "https://api.smith.langchain.com"  
os.environ["LANGCHAIN_API_KEY"] = langchain_api_key
```

</>

nome do projeto

```
from langsmith import Client  
client = Client()
```

# LangSmith

## Criação de um dataset

```
example_inputs = [  
    ("What is the largest mammal?", "The blue whale"),  
    ("What do mammals and birds have in common?", "They are both warm-blooded"),  
    ("What are reptiles known for?", "Having scales"),  
    ("What's the main characteristic of amphibians?", "They live both in water and on land"),  
]
```

definir exemplos que compõem o dataset

```
dataset = client.create_dataset(  
    dataset_name="Elementary Animal Questions",  
    description="Questions and answers about animal phylogenetics."  
)
```

criação do dataset

# LangSmith

## Criação de um dataset

```
for input_prompt, output_answer in example_inputs:  
    client.create_example(  
        inputs={"question": input_prompt},  
        outputs={"answer": output_answer},  
        dataset_id=dataset.id  
    )
```

A curved arrow pointing from the 'dataset\_id' parameter in the code to the text 'adicionar exemplos ao dataset'.

adicionar exemplos ao dataset

# LangSmith

## Avaliadores

- o LangChain oferece avaliadores que podem ser usados em cenários comuns de avaliação
- uma avaliação é uma função que recebe um conjunto de entradas/saídas oferecidas por um agente, uma chain, entre outros, e devolve uma pontuação
  - a pontuação pode ser baseada na comparação do resultado obtido e do esperado (referência)
  - existem avaliadores que não dependem de referência
- as avaliações são feitas nos datasets
- os avaliadores são úteis mas ainda estão sujeitos a erros
  - não é recomendado confiar cegamente nos resultados retornados

```
evaluators = [  
    LangChainStringEvaluator("cot_qa"),  
    LangChainStringEvaluator("labeled_criteria", config={"criteria": "relevance"}),  
    LangChainStringEvaluator("labeled_criteria", config={"criteria": "conciseness"})  
]
```

# LangSmith

## Avaliação dos resultados

- alguns dos avaliadores que podem ser usados para Q&A são:
  - **"qa"** ⇒ instrui o LLM a categorizar o resultado como "correto" ou "incorreto" baseando-se na resposta de referência
  - **"context\_qa"** ⇒ instrui o LLM a utilizar o contexto de referência para determinar a exatidão
  - **"cot\_qa"** ⇒ similar ao *context\_qa* mas usa chain of thought reasoning para determinar o veredito final
- para medir a similaridade entre uma string e uma referência:
  - **"string\_distance"** ⇒ calcula uma distância normalizada entre o texto obtido e a referência
  - **"embedding\_distance"** ⇒ calcula a distância entre os embeddings do texto obtido e a referência
  - **"exact\_match"** ⇒ procura por uma correspondência exata entre o texto obtido e a referência
- é possível customizar os avaliadores
- também é possível criar novos avaliadores
  - uso de **criteria** e **score**

# Obrigada!

Repositório com os códigos: <https://github.com/NicoleBGomes/LangChain-Presentation>