

LangChain



componentes e agentes

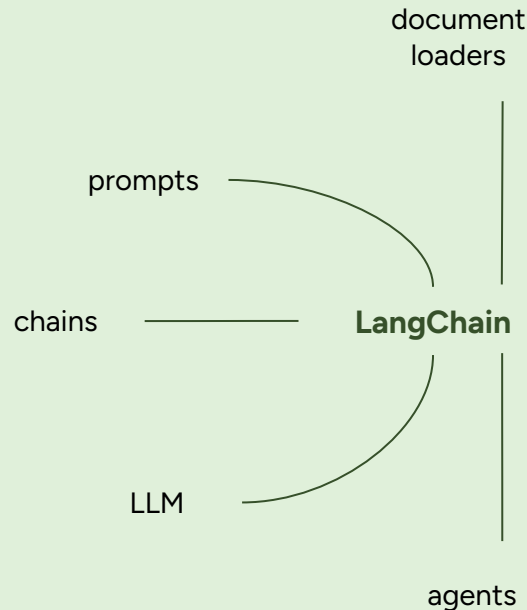
01



Sobre o LangChain

LangChain

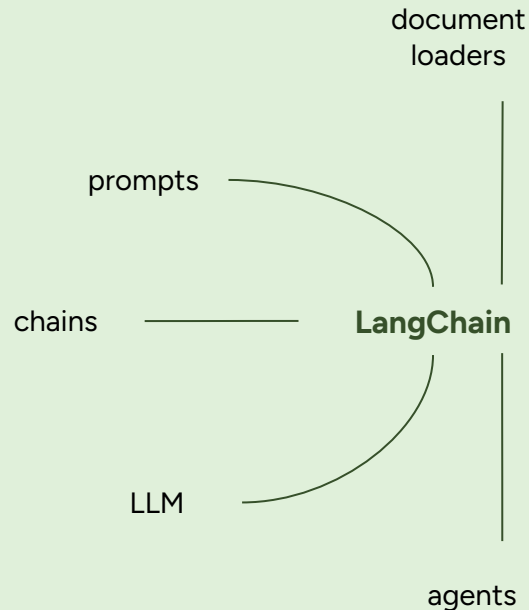
- framework de código aberto que permite criar diversas aplicações baseadas em LLMs
- criado em 2022 por Harrison Chase
 - ascensão maior em junho de 2023
 - projeto de código aberto que mais crescia no Github
 - ajudou a tornar a IA generativa mais acessível
- disponível para python e java
- possibilita 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

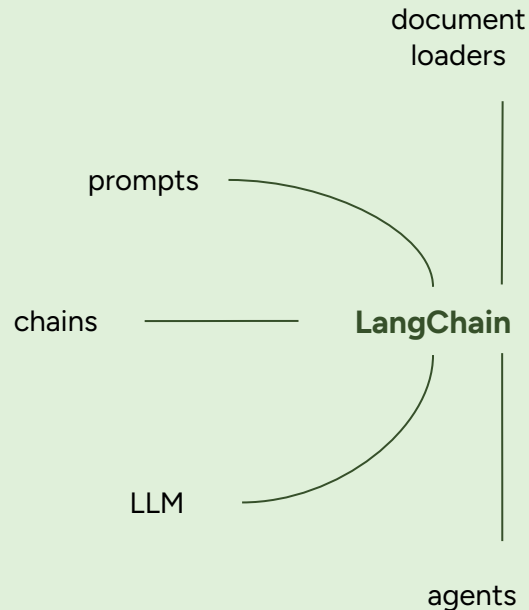
- tarefas podem precisar acessar informações contextuais específicas
 - documentação interna
 - conhecimento de domínio
- os LLMs precisam estar conectados a essas fontes externas de dados, de alguma forma
 - acesso a APIs, por exemplo
- essas integrações podem ser feitas manualmente mas o LangChain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



LangChain

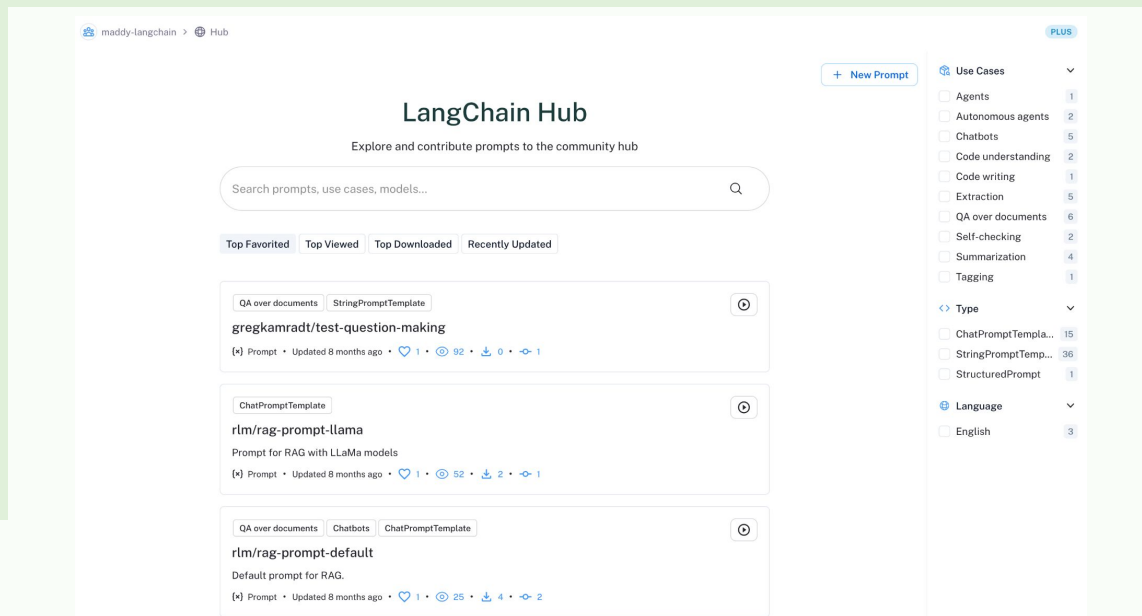
Como funciona?

- 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
- pode limitar a personalização desejada por um programador mais especializado
 - mas possibilita a experimentação para outros



LangChain Hub

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



02



Agentes

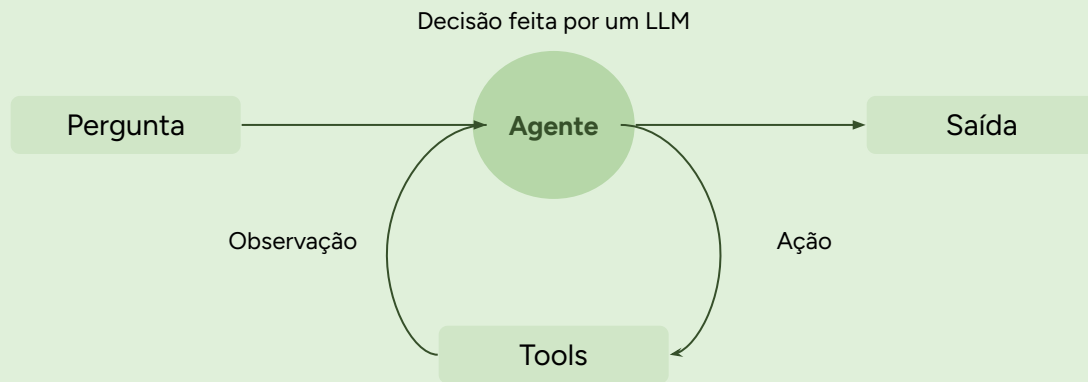
Agente

- 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



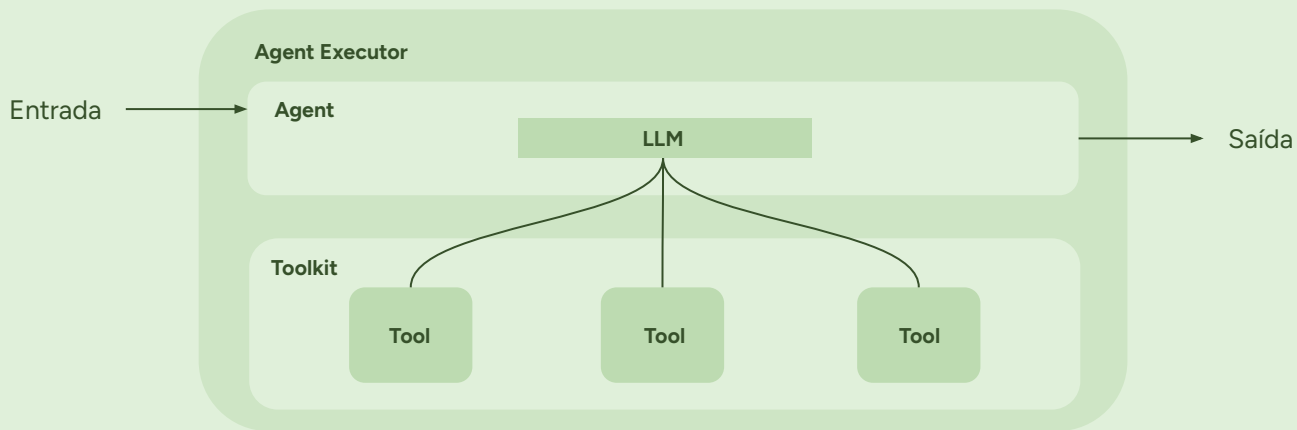
Agente

- 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

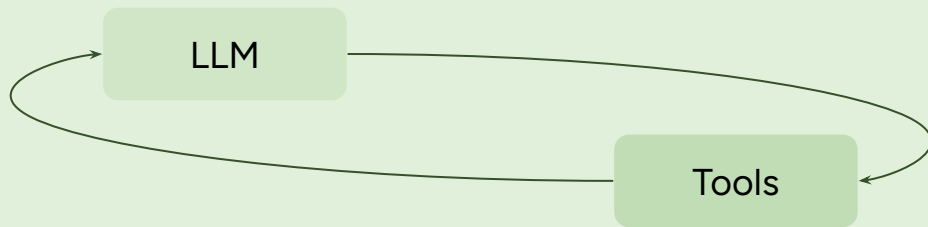
03



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."  
)
```

04

**Outras
possibilidades**



Agente

Adicionar memória ao agente

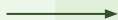
- para aplicações com aspecto conversacional
- uma das maneiras possíveis:
 - alterar o prompt do agente para que tenha uma variável para inserção do histórico: **{chat_history}**
 - uso de **ConversationBufferMemory** ➔ armazena as mensagens, que são extraídas para uma variável
 - passar a variável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

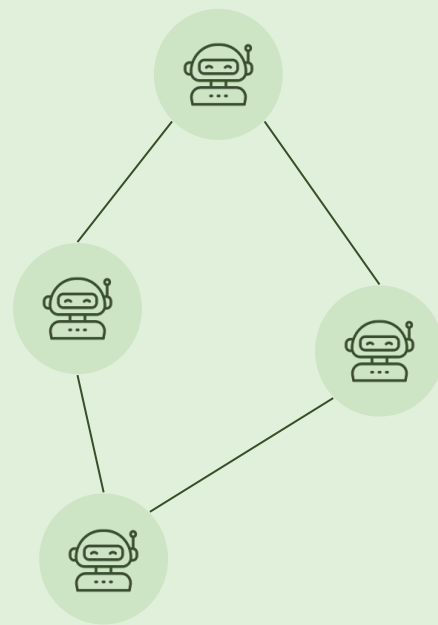
Adicionar memória ao agente

- 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)
```

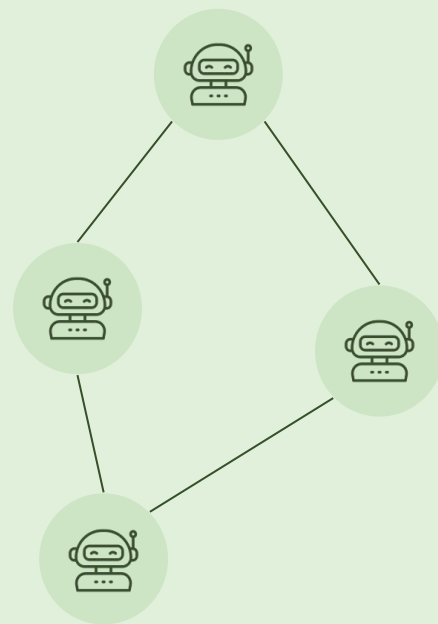

LangGraph

- 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

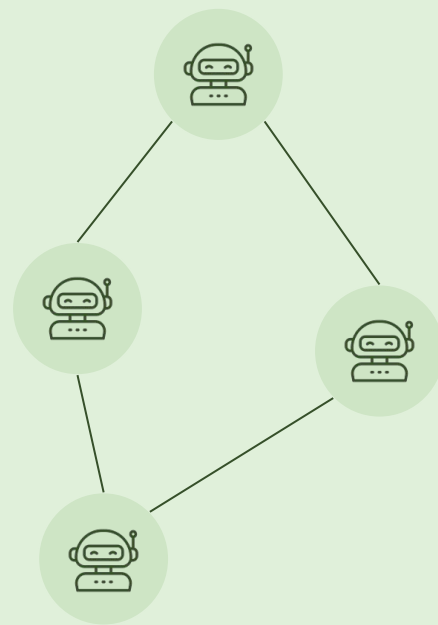
- na prática:
 - **StateGraph** ➔ classe que representa o grafo
 - para instanciá-la, é necessário passar o **state** como parâmetro, que é atualizado pelos nós do grafo

```
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?")

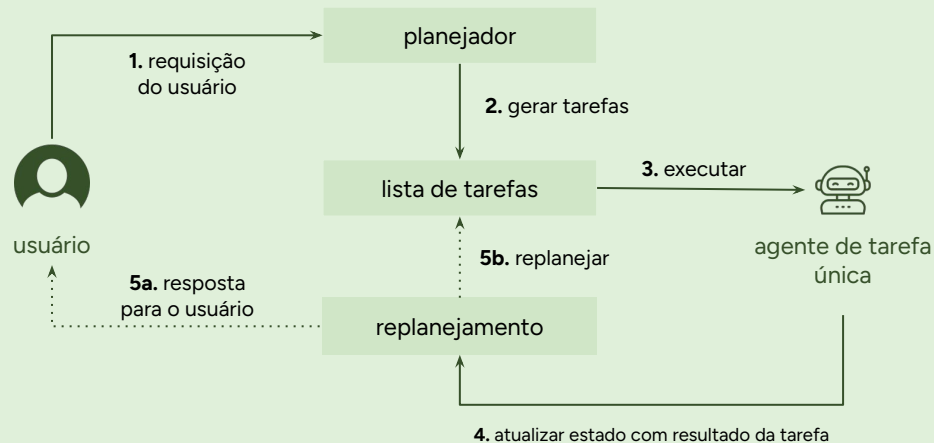
Observação: The current score is 24-21

... (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

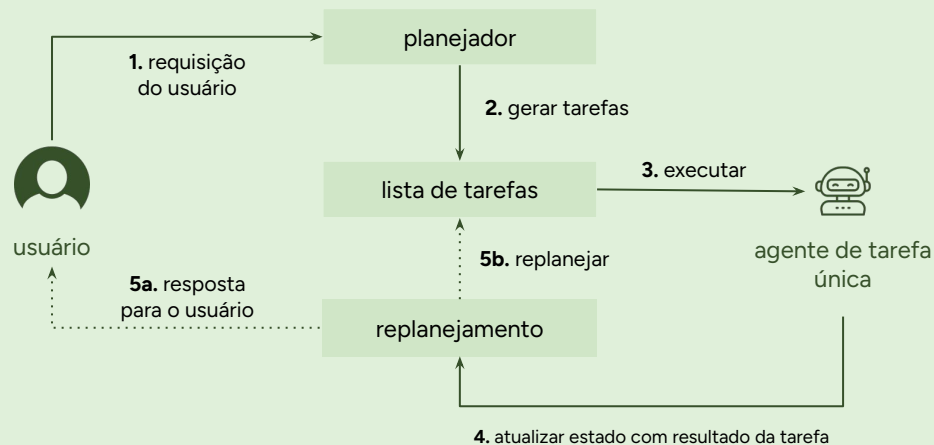
- consiste em dois componentes:
 - um **planejador** que pede para o LLM gerar um plano com múltiplos passos para realizar uma tarefa grande
 - um **executor** que recebe a entrada do usuário e uma etapa do plano, e chama uma ou mais tools para completar essa tarefa
- quando a execução termina, o agente 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

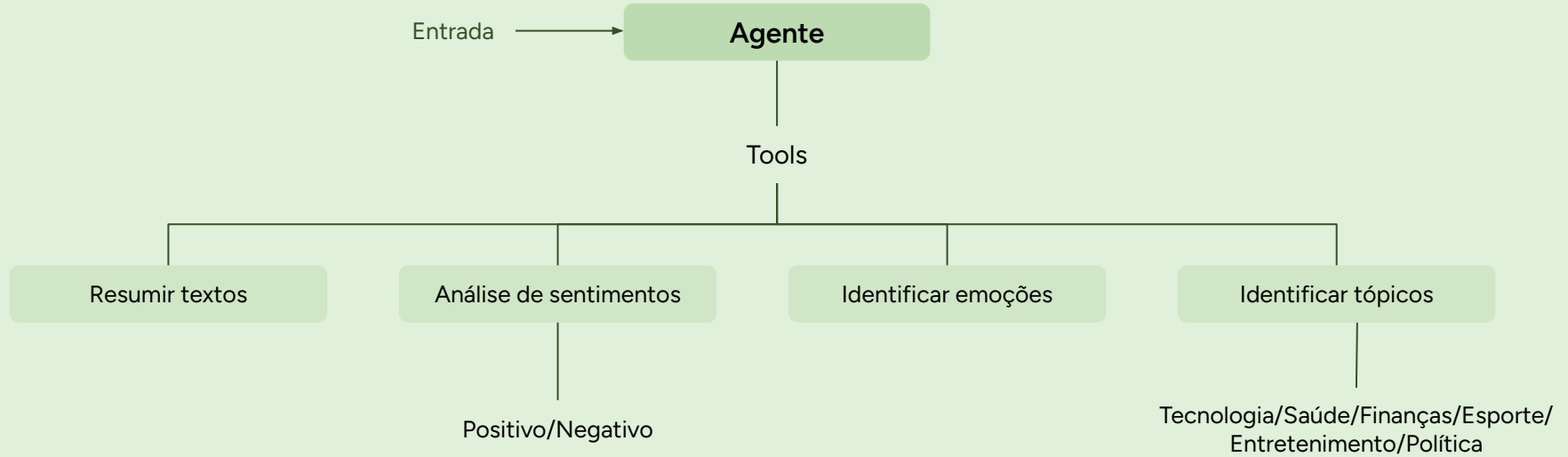


05



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

06

Componentes Operacionais



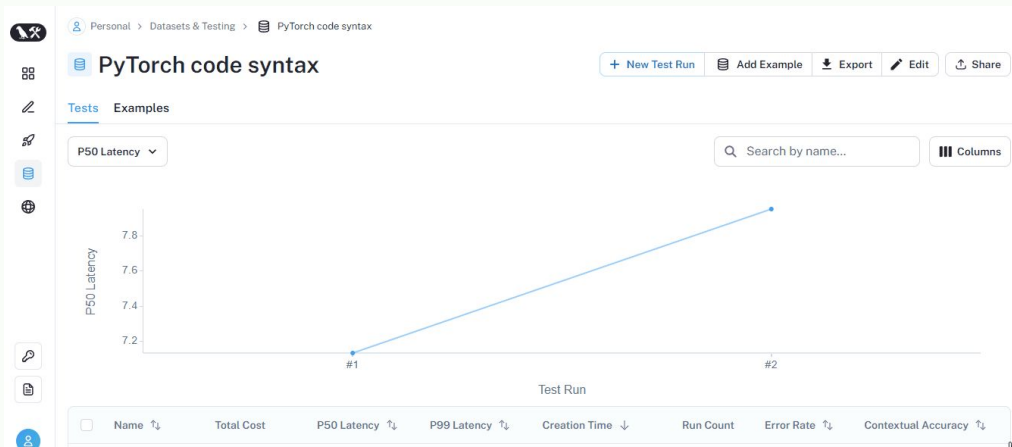
Componentes operacionais

- 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 'create_example' function call in the code block 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

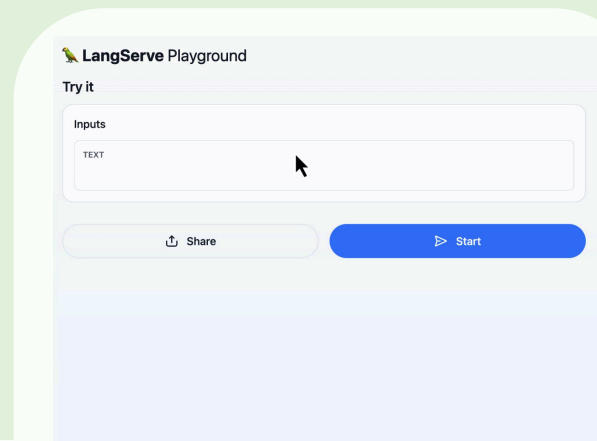
- alguns dos avaliadores que podem ser usados para Q&A são:
 - **"qa"** ⇒ instrui o LLM a categorizar o resultado como "correto" ou "incorreto" baseando-se na resposta de referência
 - **"context_qa"** ⇒ instrui o LLM a utilizar o contexto de referência para determinar a exatidão
 - **"cot_qa"** ⇒ similar 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**

LangServe

- uma maneira mais fácil de implementar os agentes ou outras aplicações criadas através do LangChain
 - ajuda a levar protótipos para a fase de produção
 - garante que sua aplicação possa lidar com múltiplas requisições ao mesmo tempo
 - oferece maneiras de ver os passos intermediários dentro da aplicação
-
- em alto nível, é necessário:
 - criar uma aplicação
 - passá-la para o LangServe
 - com algumas linhas de código, é possível ter um servidor web que atenda a requisições

LangServe

- ajuda os desenvolvedores a implementar as chains e os outras criações do LangChain, na forma de REST APIs
- biblioteca integrada com FastAPI e que usa pydantic para validação dos dados
- oferece API endpoints para métodos intrínsecos do LangChain, como *invoke*, *batch* e *stream*
- infere, automaticamente, os schemas dos inputs e outputs para os objetos LangChain
- monitoramento de performance das APIs
- playground interativo que possibilita testes e melhorias



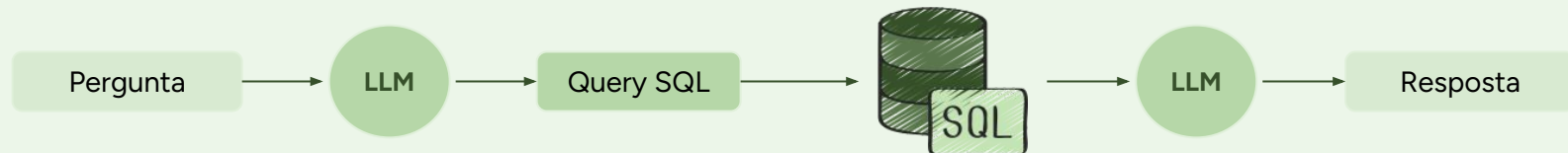
07



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 uma chain 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?"})
```

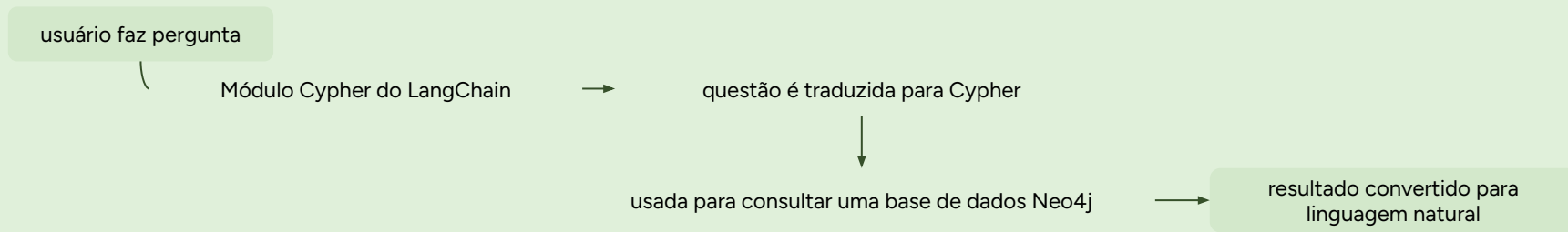
Desafios e possíveis problemas dos agentes

Fazer com que os agentes:

- usem as tools apropriadas
 - não usem nenhuma tool quando não for necessário
 - interpretem a saída do LLM para, então, invocar uma tool
 - se lembrem dos passos escolhidos
 - incorporem os resultados devolvidos pelas tools
 - continuem focados
-
- dificuldade de avaliar resultados
 - avaliar o resultado final
 - avaliar os passos intermediários

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 *metada*
- 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?")
```

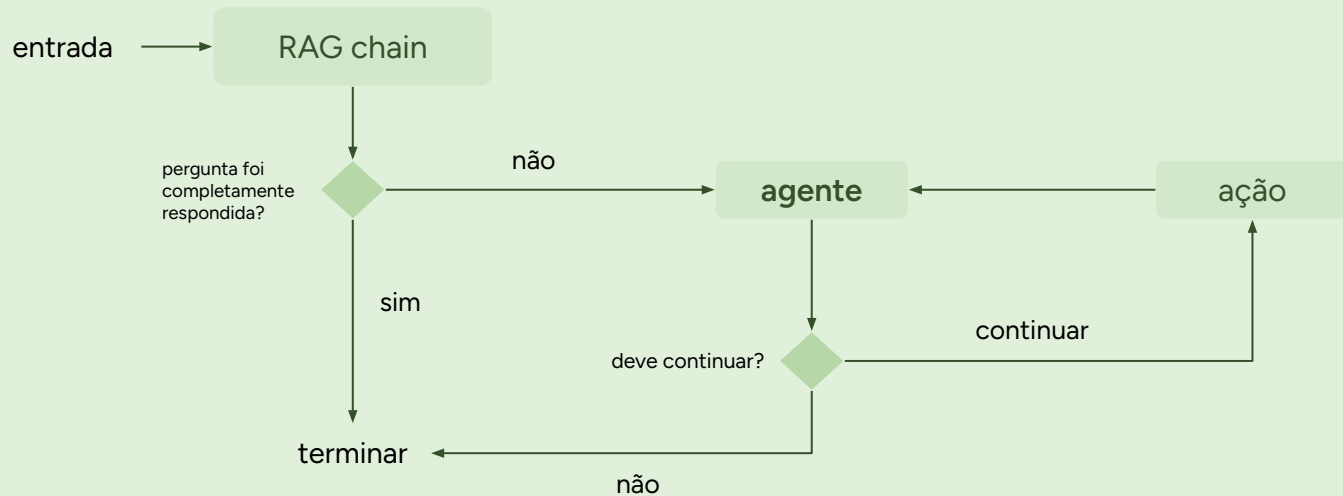
Agentic RAG

- 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



Hybrid Search

- é possível melhorar a aplicação de RAG através do uso de busca híbrida
- técnica que combina dois ou mais algoritmos de busca para melhorar a relevância dos resultados
- busca baseada em keyword + busca baseada em vetores
 - ambos os tipos têm desvantagens

Busca de keywords

sensível a erros de digitação e sinônimos

pode perder contextos importantes

Busca semântica

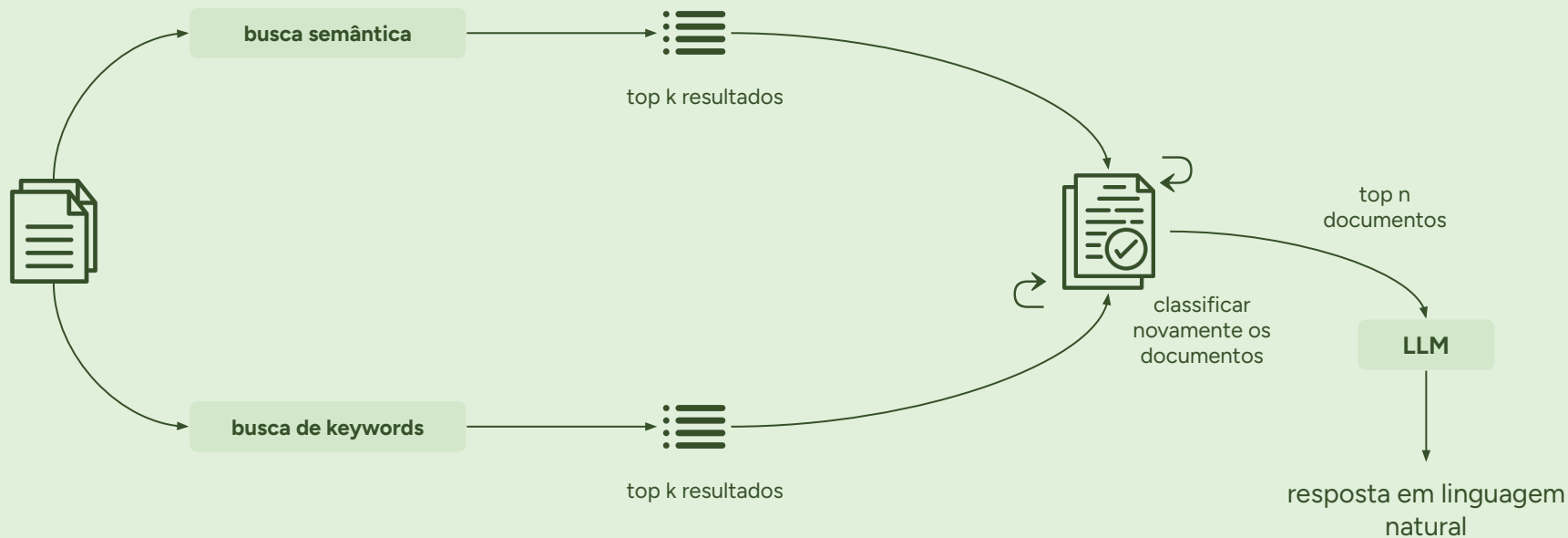
pode perder palavras-chave essenciais

apresenta diferentes resultados dependendo da qualidade dos embeddings

Hybrid Search

- **Busca baseada em keyword**
 - um algoritmo comum para gerar esse tipo de busca é o BM25
 - enfatiza a importância dos termos baseado em suas frequências
- **Busca semântica**
 - uso de vector embeddings
 - encontra os objetos mais próximos a query utilizada para a busca
- **Busca híbrida**
 - combina os resultados dos dois tipos de busca e ordena novamente os documentos retornados
 - as vantagens dos dois tipos de busca são complementares

Hybrid Search



Hybrid Search

- várias técnicas para combinar os resultados
 - os resultados recebem uma pontuação
 - as pontuações calculadas são ponderadas a partir de um parâmetro *alpha*
 - determina o peso de cada algoritmo e seu impacto na nova ordenação dos resultados
 - geralmente, assume um valor entre 0 e 1

alpha = 1: pura busca semântica
alpha = 0: pura busca de keywords

- o alpha, que controla os pesos das diferentes buscas, pode ser considerado um hiperparâmetro que precisa ser ajustado

Hybrid Search

- No LangChain:
 - é possível usar a vector store **Weaviate**

```
from langchain.retrievers.weaviate_hybrid_search import WeaviateHybridSearchRetriever

retriever = WeaviateHybridSearchRetriever(
    alpha = 0.5,
    client = client,
    index_name = "LangChain",
    text_key = "text",
    attributes = []
)
```

- usar **Ensemble Retriever**

```
from langchain.retrievers import EnsembleRetriever

ensemble_retriever = EnsembleRetriever(
    retrievers = [bm25_retriever, faiss_retriever], weights=[0.5, 0.5]
)
```

Hybrid Search

Ensemble Retriever

- busca por keywords

```
doc_list = [  
    "To know the direction, you have to look right",  
    "This is correct",  
    "You are right",  
    "Right after the meeting, we can have lunch",  
    "Turn left at the next intersection"  
]
```



```
bm25_retriever = BM25Retriever.from_texts(doc_list)  
bm25_retriever.k = 2
```



```
bm25_retriever.invoke("right")
```

Hybrid Search

Ensemble Retriever

- busca por similaridade



```
faiss_vectorstore = FAISS.from_texts(doc_list, embeddings)
faiss_retriever = faiss_vectorstore.as_retriever(search_kwargs={"k": 2})
```

```
faiss_retriever.invoke("right")
```

Hybrid Search

Ensemble Retriever



```
ensemble_retriever = EnsembleRetriever(retrievers=[bm25_retriever,  
faiss_retriever], weights=[0.5, 0.5])
```

```
docs = ensemble_retriever.invoke("right")  
docs
```

Obrigada!