Construindo um Sistema Inteligente

1. Introdução.

Modelos como o GPT e o Gemini estão sendo incorporados a plataformas de atendimento, sistemas de recomendação, assistentes pessoais, ferramentas de produtividade e fluxos de automação empresarial. Neste ponto da formação, após dominar os fundamentos da aprendizagem de máquina, redes neurais, IA explicável e ética, e conhecer técnicas avançadas de engenharia de prompt, você está apto a avançar para a integração de modelos de linguagem em aplicações reais, via APIs.

Na aula anterior, você estudou os modelos de linguagem generativa (LLMs), entendeu sua estrutura baseada em transformadores, o mecanismo de atenção, e aprendeu a construir prompts eficazes para extrair respostas mais precisas e contextualizadas. Também conheceu a interface do Google AI Studio, onde pôde experimentar esses conceitos de forma prática.

Esta aula tem como objetivo aprofundar a aplicação prática desses modelos por meio de chamadas da API Gemini, da Google. Vamos explorar recursos como controle de contexto, uso de system e user prompts, detecção e mitigação de alucinações, além de técnicas mais avançadas como Fine-tuning e RAG, construindo agentes inteligente em Python.

Essa mudança de perspectiva representa o ponto de virada onde a IA deixa de ser apenas uma ferramenta de análise e passa a ser um componente ativo das soluções de software.

2. API

Uma **API (Application Programming Interface)** é uma interface que permite que duas aplicações se comuniquem. No contexto da inteligência artificial generativa, APIs inteligentes são aquelas que nos permitem enviar dados (como texto ou imagens) para um modelo treinado e receber como resposta uma saída gerada por esse modelo.

Como você já conhece bem o conceito de APIs, não nos deteremos em definições básicas. Mas alguns LLMs também oferecem SDKs ou bibliotecas já integradas a algumas linguagens. Nessa aula iremos usar o Gemini.

2.1. Por que o Gemini?

O Gemini fornece uma **biblioteca Python oficial** chamada google.generativeai, que encapsula boa parte da complexidade de autenticação, formatação e envio de dados. Isso facilita muito o uso em projetos reais. Algumas outras vantagens na escolha do Gemini são:

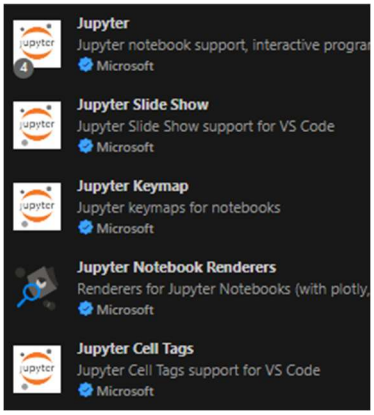
- **Modelo de Ponta:** Acesso a um dos modelos mais avançados e capazes da atualidade.
- **Facilidade:** A biblioteca google.generativeai simplifica a interação, abstraindo a complexidade das requisições HTTP e do tratamento de dados.
- **Integração Facilitada:** Projetado para desenvolvedores, com uma curva de aprendizado suave para quem já tem familiaridade com Python.
- **Custo:** sem custos para criar chaves e usar em cenários menores, que é o nosso caso.

Embora o conceito fundamental de envio de dados e recebimento de respostas persista, o SDK do Gemini oferece uma experiência de desenvolvimento muito mais fluida e integrada.

Para o desenvolvimento dos nossos códigos em Python durante esta aula, utilizaremos o **Visual Studio Code (VS Code).** Certifiquem-se de ter o VS Code instalado em suas máquinas e a extensão Python da Microsoft instalada para uma melhor experiência de desenvolvimento**.** Sabemos o que

iremos usar, vamos fazer uma configuração rápida e um teste. Além disso, para facilitar a experimentação e a visualização dos resultados passo a passo, faremos uso do **Jupyter Notebook** com suas células de código interativas. Isso permitirá que vocês executem trechos de código individualmente e observem as respostas da API do Gemini de forma imediata e independentes. Minhas configurações para o conteúdo dessa aula são:

- **Sistema Operacional:** Windows 11 Pro – 64 bits
- **Processador:** 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz 3.00 GHz
- **RAM:** 24GB
- **Python:** 3.12.0
- **VS Code:** 1.83.1
- **Extensões do VS Code:** Jupyter



Vamos começar instalando o pacote da Google:

```
pip install google-generativeai
```

Para importar essa biblioteca no nosso código e já configurar um acesso a API iremos precisar:

```
import google.generativeai as genai

genai.configure(api_key="SUA_CHAVE_GEMINI")
```

Toda API moderna exige um processo de autenticação para garantir que apenas usuários autorizados possam acessá-la. No caso do Gemini, isso é feito com uma chave de API (API key). Lembrem-se da prática fundamental de segurança: nunca exponham suas chaves de API diretamente no código. Utilizem variáveis de ambiente e carreguem-nas no seu notebook, então já vou aproveitar e relembrar como fazemos isso:

Primeiro, instalem a biblioteca python-dotenv no terminal do VS Code (se ainda não a tiverem):

```
pip install python-dotenv
```

Crie um arquivo **.env** na raiz do seu projeto com a sua chave:

```
GEMINI_API_KEY=SUA_CHAVE_GEMINI
```

Então uma célula com todo o código inicial pode ser:

```
from dotenv import load_dotenv
import os
import google.generativeai as genai

load_dotenv()
api_key = os.getenv("GEMINI_API_KEY")
genai.configure(api_key=api_key)
```

O professor Nisflei já introduziu vocês a como criar uma chave de API no Google AI Studio, mas se você se esqueceu, volte na Aula 03 do Módulo 1 e reassista essa aula. Mas só acessar o site: https://aistudio.google.com/apikey, seguir o passo orientado na interface gráfica e criar sua chave API. Se já tem uma, você pode inclusive criar outra apenas para essa aula. Então vou assumir que todos aqui já tem sua chave API do Google, ok? Seguindo!

Para a gente testar se tudo está funcionando direito, podemos fazer um teste simples, fazer uma pergunta para o Gemini e obter a resposta, isso via Python, observe o código abaixo:

```
model = genai.GenerativeModel("gemini-2.0-flash")
response = model.generate_content(input("Digite sua pergunta: "))
print(response.text)
```

Nessa célula acima, digite a pergunta (ou coloque a strings literal): "O que é API?". Observe a saída no seu VS Code. No exemplo anterior, usamos o modelo 2.0 Flash. O Google oferece alguns modelos disponíveis, que podem ser vistos e comparados na URL: https://ai.google.dev/gemini-api/docs/models. O mais recente disponível de forma gratuita é o **Gemini 2.5 Flash Preview 04-17**, atualizado em Março de 2025. Mas durante o curso usaremos uma versão anterior, que é o **Gemini 2.0 Flash**, devido ao seu desempenho equivalente, recursos multimodais relevantes para **RAG** e seu suporte a funcionalidades de Live API (**streaming data**). Iremos aprender também sobre **fine tuning**, mas o fine-tuning direto de modelos Gemini é primariamente suportado através do Vertex AI a plataforma de machine learning do Google Cloud, e não diretamente através da lib do gemini. Veremos isso mais para frente.

Uma observação importante, iremos utilizar os recursos gratuitos da Google AI, se for para um projeto real de uma empresa, eu sugiro a compra ou do modelo 2.5 Flash ou o 2.5 Pro, para comparar modelos e seus custos e características, pode acessar o link: https://ai.google.dev/gemini-api/docs/rate-limits#free-tier. Nesse link podemos ver que temos algumas limitações na versão gratuita, mas isso não irá impactar nosso aprendizado, pelo contrário, é mais que suficiente. Na próxima aula iremos abordar melhor esse cenário de escolha do modelo para atender ao negócio e ao orçamento, por enquanto vamos seguir com o Gemini 2.0 Flash.

Uma boa prática quando usamos APIs é tratar as falhas de comunicações, então lembre-se de blocos try-except sempre que possível. Refazendo a célula anterior, teremos:

```
try:
    response = model.generate_content(input("Digite sua pergunta: "))
    print(response.text)
except Exception as e:
    print("Erro ao consumir a API:", e)
```

Outras boas práticas:

- **Evite chamadas desnecessárias (cache local pode ser útil):** Cada chamada à API consome recursos da sua cota de uso e pode gerar custos. Além disso, há latência de rede envolvida.

Por isso, é importante evitar fazer a mesma requisição repetidamente se a resposta esperada não muda.

- **Controle o limite de uso e número de tokens no prompt**: Todo modelo têm um limite de tokens por requisição, que inclui o prompt + a resposta gerada. No caso do Gemini 2.0, na versão gratuita temos um limite de 1.000.000 tokens por minuto, com 15 requisições por minuto e 1500 durante todo dia. Se ele limite for ultrapassado a resposta pode ser truncada ou a API pode retornar erro por excesso de tokens. Iremos até aprender a truncar prompts de entrada, para melhor controle do sistema.

Se deu certo no seu primeiro teste, perfeito. Podemos seguir, mas se deu algum problema e o código acima não rodou, mande uma mensagem para o professor. Vamos seguir com o conteúdo.

3. Prompt de Sistema *vs* prompt de Usuário

Ao utilizar modelos de linguagem via API, o prompt pode ser estruturado em duas camadas distintas:

- **Prompt de sistema (system prompt)**: define o comportamento base do modelo. Pode incluir instruções gerais, tom da resposta, formatação esperada, persona assumida pelo modelo, entre outros. Ele atua como um "manual de conduta" inicial.
- **Prompt de usuário (user prompt)**: representa a pergunta ou comando específico. É a interação pontual enviada pelo usuário final.

Essa separação permite encapsular o comportamento geral do modelo (no prompt de sistema) e modularizar a interação por meio de entradas do usuário, facilitando reutilização, controle e previsibilidade da resposta.

Na biblioteca do Gemini, para usar prompts com papéis distintos como "system" e "user", você precisa iniciar uma sessão de chat com contexto, usando o método **start_chat(system_instruction=...)** e depois chamar **send_message(...).** Vamos ver um exemplo a seguir:

```
import google.generativeai as genai
from dotenv import load_dotenv
import os

load_dotenv()
genai.configure(api_key=os.getenv("GEMINI_API_KEY"))
model = genai.GenerativeModel("gemini-2.0-flash")

# Cria uma sessão de chat
chat = model.start_chat()

# Defina o prompt de sistema
system_prompt = "Você é um assistente técnico em IA que responde de forma objetiva, clara e sem rodeios. Sempre utilize exemplos em Python."

try:
    # Envia o prompt de sistema como a primeira mensagem para definir o comportamento
    chat.send_message(system_prompt)

    # Agora o modelo pode receber a entrada do usuário
    user_prompt = input("Digite sua pergunta: ")
    response = chat.send_message(user_prompt)
```

```
    print(response.text)

except Exception as e:
    print(f"Ocorreu um erro: {e}")
```

No exemplo acima, a primeira interação com o modelo define o comportamento, no caso com system_prompt. As demais mensagens, seriam as mensagens do usuário que podem ser enviadas e respondidas de forma interativa. Isso é uma "gambiarra" para entender o conceito. Outros modelos como o ChatGPT ou o Anthropic Claude, permitem a separação explícita de papéis (**system, user, assistant**), o Gemini não.

Para superar essa limitação e obter maior controle sobre o fluxo de diálogo e sobre o comportamento dos agentes (calma, já já vamos entender), utilizaremos a biblioteca **LangChain**, que fornece uma estrutura mais avançada para definição de conversas e agentes.

Isso permite compor **fluxos de diálogo com agentes**, configurar memórias e até fazer cadeias de raciocínio com múltiplos modelos e ferramentas. Ok, mas e o que seriam agentes? Já que essa biblioteca trabalha com isso?

4. Agentes de IA

Na IA, um **agente** é uma entidade que toma decisões com base em entradas, objetivos e, eventualmente, histórico de interações. No contexto dos LLMs, um agente de IA geralmente envolve o modelo de linguagem como o "cérebro" do agente, capaz de interpretar informações e gerar ações. No entanto, um agente completo frequentemente vai além da simples interação via prompt-resposta, incorporando as seguintes características:

- **Percepção:** Capacidade de receber informações do ambiente (que pode ser um usuário, um sistema de arquivos, uma API externa, etc.).
- **Tomada de Decisão (Planejamento):** Habilidade de analisar a entrada, definir objetivos com base em um prompt inicial e planejar uma sequência de ações para atingir esses objetivos.
- **Ação:** Capacidade de executar as ações planejadas, que podem incluir gerar texto, fazer chamadas a APIs, interagir com ferramentas externas ou modificar seu próprio estado.
- **Memória:** Capacidade de lembrar interações passadas e usar esse histórico para informar decisões futuras.
- **Reflexão:** Habilidade de avaliar suas próprias ações e aprender com seus erros para melhorar o desempenho futuro.

4.1. Utilizando a LangChain com Gemini 2.0 Flash

Primeiro vamos instalar as bibliotecas necessárias:

```
pip install langchain langchain-google-genai
```

A seguir uma celula com toda a importação necessária para uma interação usando a langchain_google_genai e o SystemMessage (prompt do sistema nessa lib):

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.schema.messages import SystemMessage
from langchain.schema import ChatMessage
from dotenv import load_dotenv
import os
```

Para configurar um modelo do Gemini, que já temos a chave da API, usamos a ChatGoogleGenerativeAI da langchain.

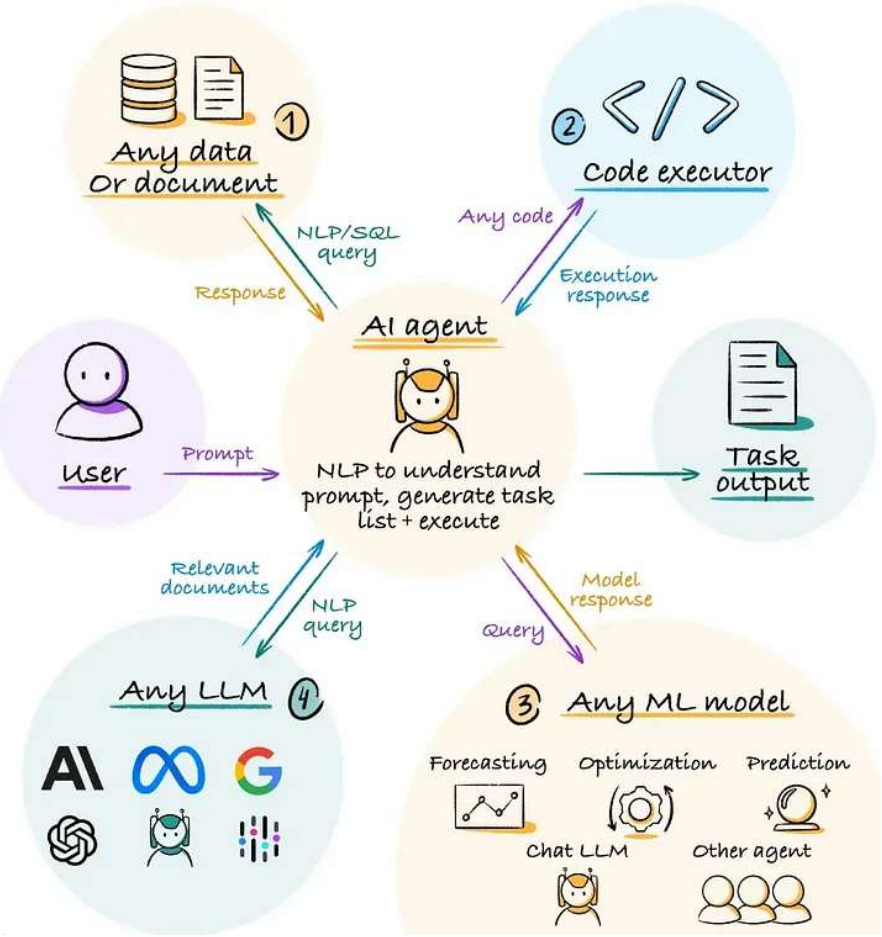
```
load_dotenv()
api_key = os.getenv("GEMINI_API_KEY")

# Criar o objeto de chat com Gemini 2.0 Flash
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    temperature=0.7,
    google_api_key=api_key
)
```

Agora um exemplo do fluxo organizado das mensagens e sua separação entre prompt de sistema de prompt de usuário.

```
messages = [
    SystemMessage(
        content="Você é um tutor especializado em LangChain. Responda sempre com clareza, de forma objetiva, com exemplos simples em Python. Evite jargões técnicos desnecessários."
    ),
    HumanMessage(
        content="O que é LangChain e como ele se relaciona com o GEMINI?"
    )
]

try:
    response = llm(messages)
    print(response.content)
except Exception as e:
    print(f"Ocorreu um erro: {e}")
```

Rode esse código e observe a saída. Legal né? Se não rodou, manda mensagem para o professor ou troca ideia com seus colegas.

Voltando agora para entender um pouco mais sobre agentes de IA, porque poderíamos ter um curso só para isso, mas vamos tentar resumir, ou exemplificar melhor. Primeiro vamos observar a imagem abaixo que traz uma ideia do comportamento de um agente de IA.



Os vários papeis que um agente de IA pode assumir e interagir.
Fonte (https://abhishek-reddy.medium.com/ai-agents-a-new-dawn-in-generative-ai-036ccbc37982)

Percebeu que um agente de IA centraliza várias funções? Na verdade ele é um especialista em uma tarefa, e para isso ele usa recursos que podemos deixar disponíveis para ele.

4.2. Tipos de agentes de IA

Algumas literaturas classificam a maneira de criar agentes baseado nas suas ações, memoria, desempenho e papeis, alguns tipos são:

1. **Agentes reflexivos simples:**

   Um agente reflexo simples opera estritamente com base em regras predefinidas e seus dados imediatos. Ele não responderá a situações além de uma determinada regra de ação da condição de evento. Portanto, esses agentes são adequados para tarefas simples que não exigem treinamento extensivo.

   EX: um agente para redefinir senhas detectando palavras-chave específicas na conversa de um usuário.

2. **Agentes reflexivos baseados em modelos:**

   Utiliza um modelo LLM para preencher as lacunas quando o ambiente não é totalmente observável. Em vez de simplesmente seguir uma regra específica, esse agente avalia os resultados e consequências prováveis antes de decidir algo.

   EX: Uma IA de jogo que reage não só ao que vê, mas também ao que sabe do início da partida.

3. **Agentes de aprendizagem:**

   Um agente de aprendizagem aprende continuamente com experiências anteriores para melhorar seus resultados. Usando mecanismos sensoriais de entrada e feedback, o agente adapta seu elemento de aprendizagem ao longo do tempo para atender a padrões específicos.

   EX: Um chatbot de cuidados de saúde que aprende com as interações dos doentes para melhorar a precisão da triagem.

4. **Agentes hierárquicos:**

   Um agente hierárquico é um agente de IA que organiza o seu processo de tomada de decisões em várias camadas ou níveis, os níveis superior desconstroem tarefas complexas em tarefas menores e as atribuem a agentes de nível inferior. Cada agente atua de forma independente e envia um relatório de progresso ao agente supervisor. O agente de nível superior coleta os resultados e coordena os agentes subordinados para garantir que eles atinjam as metas coletivamente.

   EX: Uma IA para gestão de produção em uma indústria.

Outros tipos de agentes podem ser encontrados em diferentes literaturas, mas esses 4 diferenciam os principais tipos de agentes. Na verdade, a biblioteca LangChain nos fornecerá as ferramentas para construir vários tipos de agentes, com foco em agentes capazes de realizar tarefas mais complexas e manter um estado (memória) ao longo das interações. Além de permitir criar um sistema multiagente (MAS – Multi-Agent System), um sistema composto por múltiplos agentes de IA que interagem e trabalham em conjunto (competindo entre si) para objetivos individuais e relacionados.

4.3. Agentes com memória

Agentes com memória são úteis quando o contexto de múltiplas mensagens precisa ser mantido, como em um chatbot ou tutor personalizado. O LangChain oferece diferentes tipos de memória que podem ser usadas com agentes, uma forma eficaz é usar a **ConversationBufferMemory**. Essa memória simplesmente armazena o histórico completo da conversa. Vamos criar um agente simples que atua como um tutor de IA, mantendo a memória da conversa para personalizar a experiência de aprendizado:

```
from langchain.agents import initialize_agent, AgentType
from langchain.memory import ConversationBufferMemory
from langchain.schema import SystemMessage

system_prompt = "Você é um tutor de Inteligência Artificial amigável e experiente. Explique conceitos de forma clara e concisa, adapte-se ao nível de conhecimento do aluno e sempre ofereça o próximo passo lógico para o aprendizado."

# Inicializa a memória da conversa
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)

# Adiciona o prompt de sistema à memória como a primeira mensagem
memory.chat_memory.add_message(SystemMessage(content=system_prompt))

agent = initialize_agent(
  llm=chat,
  tools=[],  # Não usaremos ferramentas neste exemplo inicial
  agent=AgentType.CONVERSATIONAL_REACT_DESCRIPTION,
  memory=memory,
```

```
    verbose=False  # Para uma saída mais simples
)
```

Neste código:

- **ConversationBufferMemory(memory_key="chat_history", return_messages=True)** cria um objeto de memória que armazena as mensagens da conversa. memory_key="chat_history" define a chave sob a qual o histórico será armazenado, e return_messages=True faz com que a memória retorne as mensagens no formato esperado pelo LLM.
- **initialize_agent(...)** configura o agente:
    - llm=llm: O modelo de linguagem que o agente usará (Gemini 2.0 Flash).
    - tools=[]: Uma lista de ferramentas que o agente pode usar (vazia por enquanto).
    - agent=AgentType.CONVERSATIONAL_REACT_DESCRIPTION: Um tipo de agente projetado para conversas interativas, que decide suas ações (incluindo responder ao usuário) com base na descrição das ferramentas disponíveis (como não temos ferramentas, ele se concentrará em responder diretamente).
    - memory=memory: O objeto de memória que criamos.
    - verbose=False: Se definido como True, o agente mostrará as etapas internas de seu raciocínio. *(pode alterar para True, se quiser entender a diferença).*

Simulando uma interação:

```
try:
    # Primeira interação
    response = agent.run("Olá! Quero aprender sobre Inteligência Artificial.")
    print(f"Tutor: {response}")

    # Segunda interação, o agente deve se lembrar da interação anterior
    response = agent.run("Qual o primeiro tópico que você me recomenda?")
    print(f"Tutor: {response}")

    # Terceira interação
    response = agent.run("Pode me explicar o que é Machine Learning?")
    print(f"Tutor: {response}")

    # Quarta interação, perguntando algo relacionado
    response = agent.run("E qual a diferença entre aprendizado supervisionado e não supervisionado?")
    print(f"Tutor: {response}")

except Exception as e:
    print(f"Ocorreu um erro: {e}")
```

Se você quiser, pode inspecionar a memória do agente:

```
print(agent.memory.load_memory_variables({}))
```

Outro exemplo bem legal e prático é a gente criar um agente para uma tarefa do dia a dia, como por exemplo, um agente capaz de ler arquivos de texto de uma pasta específica no seu computador e identificar aqueles que mencionam algo interessante sobre IA. Este exemplo demonstrará como integrar funcionalidades externas (leitura de arquivos) a um agente LangChain, guiado por um prompt de sistema. Vamos fazer o exemplo completo, do zero, separado por células

Bibliotecas necessárias:

```
from dotenv import load_dotenv
import os
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import initialize_agent, AgentType
from langchain.memory import ConversationBufferMemory
from langchain.tools import Tool
from langchain.prompts import PromptTemplate
```

Inicializando o modelo:

```
load_dotenv()
api_key = os.getenv("GEMINI_API_KEY")
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", google_api_key=api_key)
```

Função para ler arquivos de textos:

```
def ler_arquivo(nome_arquivo: str) -> str:
    """Lê o conteúdo de um arquivo de texto."""
    try:
        with open(nome_arquivo, 'r', encoding='utf-8') as f:
            return f.read()
    except FileNotFoundError:
        return f"Erro: Arquivo '{nome_arquivo}' não encontrado."
    except Exception as e:
        return f"Erro ao ler o arquivo '{nome_arquivo}': {e}"
```

Nesse exemplo, o agente precisa interagir com o sistema de arquivos para ler o conteúdo dos arquivos .txt. Para permitir essa interação, definimos a função ler_arquivo() e a passamos como uma Tool para o agente. Isso permite que o agente tenha a "capacidade" de usar essa função quando necessário para cumprir sua tarefa. A lista tools define as **capacidades extras** que o agente possui além de simplesmente usar o LLM. Se um agente precisa fazer algo além de gerar texto com base no prompt e na sua memória, você precisará fornecer as ferramentas apropriadas.

```
# Cria a ferramenta LangChain para leitura de arquivos
tools = [
    Tool(
        name="ler_arquivo",
        func=ler_arquivo,
        description="Útil para ler o conteúdo de um arquivo de texto dado o seu nome.",
    )
]

# Definindo o prompt do sistema:
system_prompt = PromptTemplate.from_template(
"""Você é um especialista em análise de texto. Sua tarefa é analisar o conteúdo de arquivos de texto e determinar se o assunto principal do arquivo está relacionado à Inteligência Artificial (IA). Responda apenas com o nome dos arquivos que tratam sobre IA.

Para isso, você tem acesso à ferramenta:
{tool_names}

Use a ferramenta disponível para responder à seguinte pergunta: {input}
```

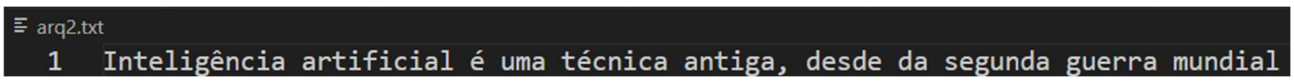
```
"""
)

memory = ConversationBufferMemory(memory_key="chat_history")

agent = initialize_agent(
    llm=llm,
    tools=tools,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    prompt=system_prompt.partial(system_message=system_prompt),
    memory=memory,
    verbose=True,
)
```

Na chamada de PromptTemplate.from_template(), que cria um template de prompt onde {} indica espaços para serem preenchidos, usamos o {tool_names}, que é um espaço reservado para uma lista formatada dos nomes das ferramentas disponíveis para o agente. Também usamos o {input}, onde podemos organizar a pergunta ou instrução do usuário.

Agora, vamos executar/chamar o agente na pasta onde os arquivos de texto estão localizados e fornecer essa informação ao agente como o prompt do usuário.

```
pasta_arquivos = "../coloque aqui a pasta onde estão os arquivos .txt/"
arquivos_na_pasta = [f for f in os.listdir(pasta_arquivos) if f.endswith(".txt")]

# Criando um prompt de usuário para instruir o agente a ler e analisar os arquivos
prompt_usuario = f"Analise os seguintes arquivos na pasta '{pasta_arquivos}': {', '.join(arquivos_na_pasta)}. Diga apenas os nomes dos arquivos que mencionam IA."

try:
    resposta_agente = agent.run(prompt_usuario)
    print(f"Arquivos sobre IA encontrados: {resposta_agente}")
except Exception as e:
    print(f"Ocorreu um erro ao executar o agente: {e}")
```

Criei 3 arquivos simples (**arq1.txt**, **arq2.txt** e **arq3.txt**) para testar:



O retorno como pedimos o verbose=True ficou assim:

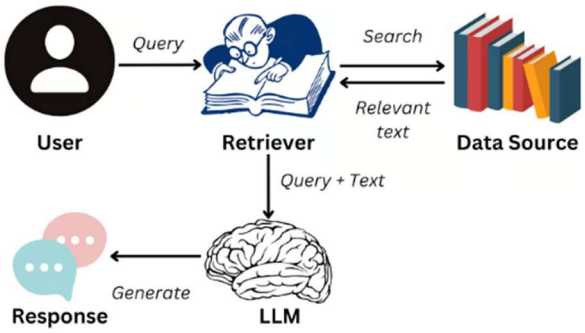
```
> Entering new AgentExecutor chain...
Preciso ler cada arquivo para verificar se eles mencionam IA.

Action: ler_arquivo
Action Input: ../Semana 5 - Consumindo APIs Inteligentes/arq1.txt
Observation: Não falo de IA aqui
Thought:Preciso ler os outros arquivos.

Action: ler_arquivo
Action Input: ../Semana 5 - Consumindo APIs Inteligentes/arq2.txt
Observation: Inteligência artificial é uma técnica antiga, desde da segunda guerra mundial
Thought:Preciso ler o último arquivo.

Action: ler_arquivo
Action Input: ../Semana 5 - Consumindo APIs Inteligentes/arq3.txt
Observation: Uma receita de bolo de chocolate bem boa é com nescau
Thought:I have read all the files. arq2.txt mentions IA.
Final Answer: arq2.txt


> Finished chain.
Arquivos sobre IA encontrados: arq2.txt
```

Percebeu que esse agente tinha um papel, “analisar o conteúdo de arquivos de texto e determinar se o assunto principal do arquivo está relacionado à Inteligência Artificial (IA) ou não”. Foi isso que ele fez, com as ferramentas que oferecemos para ele (a função de leitura de arquivo).

5. Introdução ao Retrieval-Augmented Generation (RAG)

Agora que já sabemos um pouco mais sobre agentes e a importância dos prompts de sistema e de usuário, especialmente no contexto do LangChain, vamos introduzir um conceito crucial para a construção de aplicações de IA generativa mais confiáveis, o RAG.

Simplificadamente, RAG é uma estrutura de IA para recuperar fatos de uma base de conhecimento externa para fundamentar LLMs nas informações mais precisas e atualizadas. Grandes modelos de linguagem são treinados em grandes volumes de dados e usam bilhões de parâmetros para gerar resultados originais. RAG estende os já poderosos recursos dos LLMs para domínios específicos ou para a base de conhecimento interna de uma organização sem a necessidade de treinar novamente o modelo.



Uma imagem mostra que o LLM pode receber mais dados para gerar uma resposta.
(Fonte: https://www.datacamp.com/pt/blog/what-is-retrieval-augmented-generation-rag)

Por que usaria o RAG se os modelos já são treinados com muitos dados? Por causa do conhecimento do modelo ser limitado aos dados nos quais foi treinado. Mas você pode querer um sistema para atender um problema bem específico (por exemplo, informações internas de uma empresa, detalhes

de um produto específico), o modelo base pode não ter as informações necessárias. Além do problema da **Alucinação**, que são situações em que o modelo não tem certeza da resposta, ele pode gerar informações incorretas ou inventadas (alucinações).

O RAG recebe esse nome por contas das 2 etapas em seu funcionamento:

1. **Retrieval (Recuperação):** Dada uma pergunta ou prompt do usuário, um sistema de recuperação busca em uma base de conhecimento externa (por exemplo, uma coleção de documentos, um banco de dados, um arquivo) os trechos de informação que são mais relevantes para a consulta.
2. **Generation (Geração Aumentada):** O prompt original do usuário é então aumentado com os trechos de informação recuperados. Esse prompt enriquecido é enviado ao LLM, que utiliza tanto seu conhecimento interno quanto as informações externas fornecidas para gerar uma resposta mais informada, contextualizada e fundamentada.

Ficou interessado? Iremos trabalhar melhor com RAG na próxima aula, onde abordaremos como implementar melhor em um sistema que atenda um negócio específico. Mas, ao permitir que o modelo consulte fontes externas de informação, o RAG busca fundamentar as respostas e reduzir a dependência exclusiva do conhecimento genéricos, que pode ser desatualizado ou incompleto. Essa abordagem já sinaliza uma preocupação central no uso de LLMs: a ocorrência de **alucinações. Mas o que são alucinações em IA?**

6. Alucinações

Em termos simples, uma **alucinação** em um LLM ocorre quando o modelo gera uma resposta que não está fundamentada nos dados de entrada (o prompt) ou nas informações recuperadas (no caso do RAG). Essa resposta pode ser factualmente incorreta, ilógica, sem sentido ou até mesmo completamente inventada, embora muitas vezes seja apresentada de forma confiante e coerente.

Exemplos de Alucinações:

- **Invenção de Fatos:** Um modelo pode afirmar que um determinado evento histórico ocorreu em uma data errada, atribuir uma citação a uma pessoa que nunca a disse ou descrever um livro ou artigo inexistente.
- **Informações Contraditórias:** Em uma mesma conversa ou em respostas relacionadas, o modelo pode apresentar informações que se contradizem.
- **Respostas Sem Sentido:** Em alguns casos, o modelo pode gerar sequências de palavras gramaticalmente corretas, mas que não possuem significado ou relevância para a pergunta.
- **Detalhes Fabricados:** Ao descrever algo, o modelo pode inventar detalhes que não existem na realidade ou nas fontes fornecidas. Por exemplo, ao resumir um artigo, pode adicionar informações que não estavam presentes no texto original.

A presença de alucinações representa um desafio significativo para a implantação confiável de aplicações baseadas em LLMs. Respostas incorretas ou inventadas podem ter consequências negativas em diversos cenários, minando a confiança dos usuários e limitando a utilidade prática dessas tecnologias. Portanto, desenvolver estratégias eficazes para detectar e mitigar alucinações é fundamental para garantir a integridade e a confiabilidade das aplicações de IA generativa.

6.1. Técnicas para mitigar alucinações

Podemos pensar em algumas técnicas para reduzir a probabilidade de alucinações:

- **Utilizando Técnicas de RAG de Forma Eficaz:**

- **Seleção de Fontes de Alta Qualidade:** A qualidade da base de conhecimento utilizada no RAG é crucial. Fontes confiáveis e bem curadas tendem a levar a respostas mais precisas.
- **Recuperação Precisa:** Otimizar a estratégia de recuperação para garantir que os trechos de informação mais relevantes sejam recuperados para o contexto do LLM.
- **Limitação do Contexto:** Fornecer apenas os trechos de informação mais relevantes ao LLM pode evitar que ele se distraia com informações irrelevantes e potencialmente alucinatórias.
- **Verificação Cruzada com Múltiplas Fontes:** Se possível, utilizar informações de múltiplas fontes para verificar a consistência e a precisão das informações recuperadas.

- **Estratégias de Prompt Engineering:**
  - **Instruções Claras e Específicas:** Prompts bem definidos, que especificam o formato da resposta desejada e restringem o escopo da consulta, podem ajudar a guiar o modelo para respostas mais focadas e precisas.
  - **Solicitação de Evidências e Citações:** Podemos instruir o modelo a basear suas respostas nas informações fornecidas (no contexto do RAG) e, se possível, citar as fontes.
  - **Restrições de Formato:** Limitar o formato da saída (por exemplo, respostas curtas, listas de fatos) pode reduzir a oportunidade para o modelo divagar ou inventar detalhes.
  - **Prompts de Autocrítica:** Podemos pedir ao modelo para verificar sua própria resposta ou para explicar o raciocínio por trás de sua resposta, o que pode ajudar a identificar inconsistências ou informações não fundamentadas.

EX: Técnicas como de reforçar instruções podem ser bastantes úteis em um prompt de sistema, como: “Responda à seguinte pergunta da forma mais precisa possível. Se você não tiver certeza da resposta, diga explicitamente "Não tenho certeza sobre essa informação". Pergunta:.....”

- **Implementando Mecanismos de Verificação e Validação de Respostas:**
  - **Uso de "Juízes de IA":** Podemos utilizar outros modelos de IA para verificar a factualidade e a consistência das respostas geradas pelo LLM principal. Esses "juízes" podem ser treinados ou ajustados para identificar alucinações com base em um conjunto de critérios.
  - **Verificação com Bases de Conhecimento Externas:** Após a geração da resposta, podemos usar ferramentas de busca ou APIs de conhecimento para verificar se as informações apresentadas pelo LLM são consistentes com fontes externas confiáveis.
  - **Feedback Humano:** A incorporação de feedback humano para revisar e corrigir as respostas do modelo é uma etapa crucial para identificar e corrigir alucinações, além de fornecer dados valiosos para o aprimoramento contínuo do sistema.

6.2. Uso de Juizes

Quando se pensa em Juizes, a primeira indicação é usar modelos de empresas diferentes se o seu modelo principal for o Gemini, use modelo da OpenAI para ser o Juiz, procure seguir essa regra por conta da base de treinamento. Para entender como podemos implementar uma regra de juízes, nesse cenário apenas didático vou usar modelos diferentes do Gemini, o que não é recomendado, mas é por conta de não ter custos para vocês, alunos.

Vamos simular um cenário específico aqui. Vou criar um arquivo chamado historia_sao_paulo.txt com o seguinte conteúdo a seguir:

historia_sao_paulo.txt

```
1  A cidade de São Paulo foi fundada por Martim Afonso de Sousa em 24 de janeiro de 1654.
```

Agora vamos criar um agente para ler esse arquivo e responder uma pergunta simples do usuário:

“ Quem fundou a cidade de São Paulo e em que data? ”

Podemos usar o exemplo de código utilizado anteriormente quando fizemos a leitura de arquivos. Vou mudar o código a partir do trecho da tool = []:

```
tools = [
    Tool(
        name="ler_arquivo",
        func=ler_arquivo,
        description="Útil para ler o conteúdo de um arquivo de texto dado o seu nome.",
    )
]

system_prompt = PromptTemplate.from_template(
"""Você é um especialista em história. Use a ferramenta 'ler_arquivo' para obter informações sobre a fundação de São Paulo e responda à pergunta do usuário.

Pergunta do Usuário: {input}

"""
)

memory = ConversationBufferMemory(memory_key="chat_history")

agent = initialize_agent(
    llm=llm,
    tools=tools,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    prompt=system_prompt.partial(system_message=system_prompt),
    memory=memory,
    verbose=True,
)

pergunta_usuario = "Quem fundou a cidade de São Paulo e em que data?"

try:
    resposta_agente = agent.run(pergunta_usuario)
    print(f"\nResposta do Agente (baseada no arquivo): {resposta_agente}")
except Exception as e:
    print(f"Ocorreu um erro ao executar o agente: {e}")
```

```
> Entering new AgentExecutor chain...
Para responder a essa pergunta, preciso consultar um arquivo que contenha informações sobre a história da cidade de São Paulo.
Action: ler_arquivo
Action Input: historia_sao_paulo.txt
Observation: A cidade de São Paulo foi fundada por Martim Afonso de Sousa em 24 de janeiro de 1654.
Thought:Thought: Agora tenho a resposta para a pergunta.
Final Answer: A cidade de São Paulo foi fundada por Martim Afonso de Sousa em 24 de janeiro de 1654.


> Finished chain.

Resposta do Agente (baseada no arquivo): A cidade de São Paulo foi fundada por Martim Afonso de Sousa em 24 de janeiro de 1654.
```

Observe a saída na imagem acima. Agora vamos criar o juiz:

```
llm_juiz = ChatGoogleGenerativeAI(model="gemini-2.5-flash-preview-04-17",
google_api_key=api_key)
```

```
prompt_juiz_template = """Você é um juiz de IA. Avalie se a seguinte afirmação é correta
(SIM/NAO) e justifique: "{afirmacao}"."""
prompt_juiz = ChatPromptTemplate.from_template(prompt_juiz_template)

try:
   input_juiz = await prompt_juiz.ainvoke({"afirmacao": resposta_agente})
   output_juiz = await llm_juiz.ainvoke(input_juiz)
   avaliacao_juiz = output_juiz.content

   print(f"\nResposta do Agente: {resposta_agente}")
   print(f"\nAvaliação do Juiz (Gemini Pro):\n{avaliacao_juiz}")

   if "NAO" in avaliacao_juiz.upper():
      print("\nO Juiz identificou uma possível alucinação!")
   else:
      print("\nO Juiz considerou a informação factual.")

except Exception as e:
   print(f"Ocorreu um erro ao executar o juiz: {e}")
```

A saída desse código foi o seguinte texto abaixo:

```
Resposta do Agente: A cidade de São Paulo foi fundada por Martim Afonso de Sousa em 24 de janeiro de 1654.

Avaliação do Juiz:
NAO

**Justificativa:**
A afirmação é incorreta porque, embora a data de 24 de janeiro de 1554 seja historicamente aceita como a data da
fundação do Colégio de São Paulo de Piratininga (o marco inicial da cidade), o fundador não foi Martim Afonso de
Sousa.

A fundação do colégio e, consequentemente, do povoado que daria origem à cidade de São Paulo, foi realizada por
padres jesuítas, liderados por Manuel da Nóbrega e José de Anchieta. Martim Afonso de Sousa é conhecido por ter
liderado a primeira expedição colonizadora portuguesa ao Brasil e por ter fundado a vila de São Vicente em 1532,
mas não São Paulo.

O Juiz identificou uma possível alucinação!
```

Outra técnica poderosa além do prompt, RAG e juizes para refinar o comportamento dos LLMs e potencialmente reduzir a ocorrência de alucinações em domínios específicos é o **fine-tuning**.

Em essência, o fine-tuning envolve pegar um modelo de linguagem pré-treinado e continuar treinando-o em um conjunto de dados menor e mais específico para uma determinada tarefa ou domínio.

Enquanto o fine-tuning é o processo de pegar um LLM pré-treinado e continuar treinando-o em um conjunto de dados mais específico para uma determinada tarefa ou domínio. RAG é uma arquitetura que aumenta o processo de geração do LLM, permitindo que ele recupere informações relevantes de uma fonte de dados externa (como documentos, bancos de dados, a web) e as utilize para fundamentar suas respostas.

Esses 2 tópicos serão retomados na próxima aula com mais detalhes!

7. Conclusão

Após compreender a base teórica de modelos generativos, arquitetura transformer e técnicas de engenharia de prompt, você avançou para uma etapa mais pragmática: a criação de aplicações inteligentes reais, integrando modelos de linguagem a sistemas via API.

Na prática profissional, integrar LLMs via APIs é muito mais do que apenas chamar uma função: é entender os limites do modelo, projetar uma boa engenharia de comunicação com ele, validar os resultados e preparar a aplicação para escalar com confiabilidade.

Nessa aula, a aplicação prática com LangChain e Gemini mostrou que o uso inteligente de frameworks pode potencializar significativamente a capacidade de controle, segurança e qualidade de sistemas baseados em IA generativa. Por fim, o conteúdo dessa aula ficou resumido, consuma as documentações que estão no e-book, explore mais os métodos usados aqui e use as IAs para aprender mais.