Criando Agentes com Agno

Asimov Academy

ASIMOV

Conteúdo

01. O que o Agno tem de especial?

O Agno é um framework dedicado à criação de agentes de inteligência artificial utilizando Python. Ele foi desenvolvido para simplificar o trabalho de quem deseja construir agentes capazes de interagir, buscar informações, executar tarefas automatizadas e integrar diferentes sistemas. Apesar de ainda não ser o framework mais conhecido da área, o Agno vem conquistando cada vez mais espaço na comunidade graças à sua proposta de oferecer um ambiente prático, integrado e com foco em produtividade.

Enquanto outros frameworks, como LangChain, LangGraph, RayAI e soluções no-code (como N8n, Make e LangFlow), também propõem facilitar o desenvolvimento de agentes, muitos deles acabam exigindo um grande volume de código ou demandam diversas configurações para alcançar resultados mais complexos. O Agno se diferencia principalmente por entregar uma quantidade considerável de ferramentas prontas, que podem ser utilizadas quase imediatamente, e por valorizar uma curva de aprendizado rápida, sem abrir mão da performance.

Funcionalidades e diferenciais do Agno

Um dos principais diferenciais do Agno é a integração com uma grande variedade de provedores de modelos de linguagem (LLMs), permitindo que o desenvolvedor escolha com facilidade qual modelo utilizar em cada projeto ou até mesmo combine múltiplos modelos em um mesmo agente. Entre os provedores suportados estão OpenAI, Ollama, Mistral, Gemini, DeepSeek e Grok.

Além disso, o Agno disponibiliza diversas ferramentas nativas para equipar os agentes, facilitando desde a pesquisa na web até a automação de tarefas em redes sociais (como WhatsApp, Slack, Discord, e-mail e Gmail). Também oferece suporte avançado para extração de informações em sites, manipulação de bases de dados (SQL, Postgres, arquivos CSV) e execução de comandos ou scripts Python diretamente a partir do agente, incluindo interação com arquivos locais e terminal.

Outra característica importante é a facilidade para criar e integrar ferramentas personalizadas, tornando possível adaptar o agente a qualquer necessidade específica do projeto. Para quem precisa registrar históricos de conversas, o Agno inclui um módulo de storage que permite salvar interações em bancos de dados locais de maneira simples e eficiente.

A arquitetura do Agno também contempla recursos de memória, possibilitando que o agente “lembre” pontos-chave de interações anteriores, e inclui suporte a módulos de reasoning (raciocínio), bases de conhecimento e sistemas multiagentes – estes, chamados de “times” no próprio framework.

Exemplo prático

A proposta do Agno de simplificar a construção de agentes fica evidente no código. Com poucas linhas, já é possível criar um agente funcional:

```
from agno.agent import Agent
from agno.tools.tavily import TavilyTools

agent = Agent(tools=[TavilyTools()],
              show_tool_calls=True)

agent.print_response("Search tavily for 'language models'", markdown=True)
```

Conclusão

O Agno representa uma evolução significativa no desenvolvimento de agentes de inteligência artificial em Python. Sua proposta alia praticidade, variedade de integrações e foco em produtividade, atendendo tanto iniciantes quanto desenvolvedores experientes. Com ferramentas prontas, suporte a múltiplos provedores e módulos avançados como memória, reasoning e multiagentes, o framework torna possível criar sistemas sofisticados com simplicidade e clareza no código.

A cada capítulo, iremos nos aprofundar no uso dos principais recursos do Agno, oferecendo exemplos práticos e orientando a construção de agentes personalizados para diferentes contextos. Assim, o estudante estará apto a aproveitar todo o potencial da plataforma e criar soluções de IA avançadas de forma acessível e eficiente.

02. O que são agentes de IA?

Antes de iniciar a parte prática da construção de agentes com Agno, é fundamental garantir a compreensão dos conceitos que fundamentam a área. Este módulo aborda, de forma breve e teórica, a definição e os elementos essenciais de um agente de IA, preparando o terreno para os próximos passos práticos do curso.

O que define um agente de IA?

O termo “agente de IA” ou “AI Agent” tem ganhado destaque nos últimos anos. Embora seja frequentemente mencionado como parte do futuro da inteligência artificial, é importante entender o que realmente caracteriza um agente.

Para ilustrar a diferença entre um modelo tradicional e um agente, considere o exemplo do uso do ChatGPT para solicitar a redação de um e-mail:

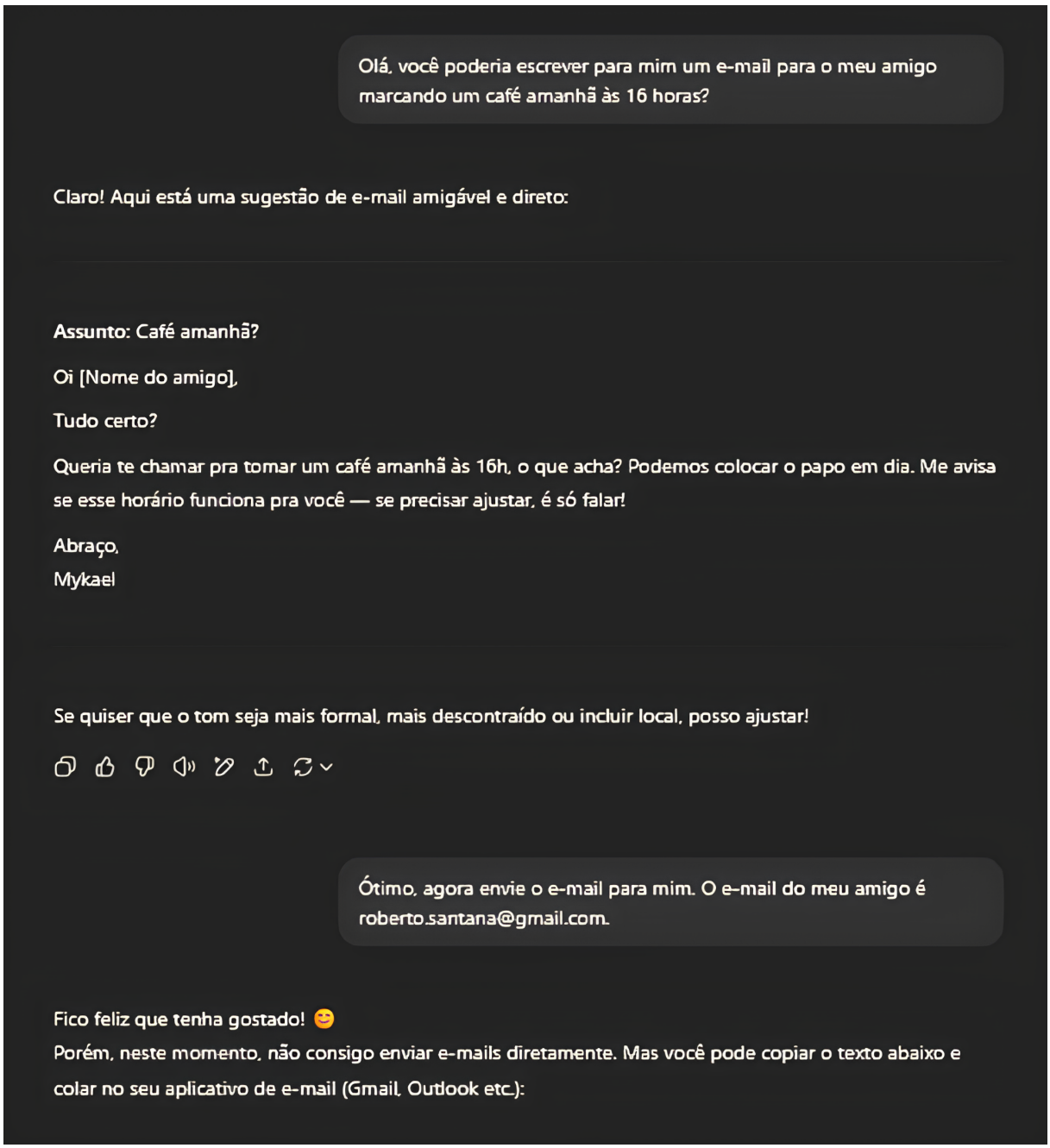

Figura 1: Exemplo diálogo com o ChatGPT

> O ChatGPT não conseguirá executar essa ação, pois está limitado apenas à geração de texto, sem acesso a ferramentas externas.

A principal limitação dos modelos tradicionais está justamente na impossibilidade de executar tarefas no mundo real, como enviar e-mails, controlar dispositivos ou acessar bancos de dados.

O papel das ferramentas nos agentes

A diferença fundamental de um agente de IA está na capacidade de interagir com o ambiente externo através de ferramentas. Isso pode incluir:

- Acesso a bancos de dados
- Envio de e-mails
- Integração com automação residencial
- Manipulação de arquivos ou informações em tempo real

Por exemplo, um agente construído com Agno pode ser equipado com uma ferramenta que acessa um banco de dados de roteiros de vídeo. Ao solicitar uma lista de criadores cadastrados, o agente chama uma função específica (“List Creators”), executa a busca e retorna as informações, automatizando uma tarefa real e personalizada.

Se o agente for configurado com ferramentas de automação residencial, pode executar comandos físicos, como desligar a luz de um cômodo, mediante uma simples solicitação em linguagem natural.

A presença dessas ferramentas é o que transforma um modelo de linguagem tradicional em um agente de IA.

Definição de agente de IA

De forma objetiva:

> **Um agente de IA é uma LLM (Large Language Model) equipada com ferramentas e rodando dentro de um loop, capaz de executar ações no ambiente ao receber comandos em linguagem natural.**

Conclusão

A evolução dos agentes de IA amplia o papel dos modelos de linguagem, permitindo que eles não apenas compreendam e gerem texto, mas também executem tarefas automatizadas no mundo real. A integração de ferramentas transforma o agente em uma peça ativa no ecossistema digital, com potencial para impactar desde automações simples até processos complexos. Entender essa diferença é fundamental para aproveitar todo o poder das plataformas modernas de agentes, como o Agno.

03. Como um agente executa tarefas?

Após compreender o conceito de agentes de IA, surge uma questão essencial: como esses agentes realmente executam tarefas? Afinal, um modelo de linguagem tradicional (LLM) é, por natureza, apenas uma grande rede neural capaz de gerar texto, sem capacidade direta de interação com o mundo externo.

Limitação dos modelos de linguagem

Quando um modelo como o Llama ou o GPT está rodando localmente ou em servidores, sua função básica é receber entradas de texto e devolver saídas em texto. Esses modelos não possuem, por si só, a capacidade de clicar em telas, executar comandos, controlar sistemas ou realizar ações no ambiente.

Portanto, toda execução de tarefa por um agente de IA depende de uma ponte entre o modelo de linguagem e o ambiente externo.

O papel do framework na execução

A mudança fundamental surgiu quando frameworks e APIs, como o próprio Agno ou as APIs da OpenAI (a partir de 2023, com o lançamento do recurso Function Calling), passaram a permitir que o modelo de linguagem fosse informado sobre quais funções estavam disponíveis para ele executar.

No prompt enviado ao modelo, são listadas as funções e instruções de como solicitar sua execução. O modelo, ao reconhecer que determinada tarefa não pode ser realizada apenas com texto, gera uma mensagem estruturada (por exemplo, em formato JSON) indicando qual função deseja chamar e quais argumentos devem ser utilizados.

O framework (no caso, o Agno) interpreta essa mensagem, executa a função requisitada localmente (ou na nuvem), obtém o resultado e então retorna essa resposta ao modelo de linguagem, que pode processar, formatar e apresentar ao usuário.

Exemplo prático

- O usuário solicita ao agente:
  “Liste para mim todos os criadores de conteúdo do banco de dados.”
- O modelo de linguagem, ao receber o pedido, reconhece que não consegue realizar sozinho a consulta. Em vez disso, ele gera uma mensagem estruturada solicitando ao framework (Agno) a execução de uma função específica, como `List Creators`.

- O Agno executa a função, busca os dados no banco de dados e retorna a resposta ao modelo de linguagem.
- O modelo de linguagem, agora com acesso ao resultado, pode processar e apresentar a informação ao usuário em formato de texto compreensível.

Esse processo evidencia que a execução real das tarefas parte sempre do framework ou aplicação que "escuta" as intenções do modelo e executa, de fato, as funções necessárias.

Conclusão

O modelo de linguagem, por si só, não executa tarefas no mundo real. Sua função é reconhecer intenções, interpretar comandos e gerar mensagens estruturadas para que o framework responsável (como o Agno) realize as ações necessárias. A integração entre modelo, ferramentas e aplicação permite que agentes de IA sejam capazes de realizar tarefas automatizadas, trazendo inteligência e autonomia a diferentes tipos de sistemas.

04. Agno ou LangChain?

Ao estudar agentes de IA em Python, é comum se deparar com a dúvida: qual framework utilizar? Entre as principais opções do mercado, Agno e LangChain se destacam por serem bastante completos e populares, mas atendem a necessidades diferentes.

Comparando Agno e LangChain

O LangChain é hoje a biblioteca mais conhecida para construção de aplicações e agentes com LLMs. Seu ponto forte está no nível de customização: praticamente todos os componentes e comportamentos podem ser definidos pelo desenvolvedor, possibilitando a criação de fluxos altamente personalizados. No entanto, essa flexibilidade traz também maior complexidade, exigindo mais conhecimento técnico e tempo para configurar funcionalidades como memória, gestão de storage e colaboração entre múltiplos agentes.

Por outro lado, o Agno foi desenvolvido com foco em produtividade e simplicidade. Muitas das principais funcionalidades para construção de agentes já vêm prontas, como gerenciamento de memória, execução de funções, lógica de colaboração entre agentes e sistemas multiagentes. Isso permite que o desenvolvedor comece rapidamente a construir seus projetos, gastando menos tempo com configurações técnicas e mais tempo naquilo que realmente impacta o resultado final: a elaboração dos prompts e o design das interações.

Pontos-chave:

- **Agno**
  - Vantagem: Grande parte dos recursos já implementados, velocidade de desenvolvimento, menor preocupação com detalhes técnicos.
  - Limitação: Menor flexibilidade para customizações profundas (depende das escolhas feitas pelos desenvolvedores do framework).
- **LangChain**
  - Vantagem: Alto grau de personalização e controle sobre todos os aspectos do agente e do fluxo de execução.
  - Limitação: Exige mais trabalho manual para configurar cada etapa do agente, aumentando a curva de aprendizado e o tempo de desenvolvimento.

Qual escolher?

A escolha depende do seu objetivo:

- Se a prioridade é construir agentes rapidamente, com o menor atrito possível, o Agno é recomendado. Ele permite focar naquilo que mais importa para a maioria dos casos: a qualidade do prompt e o design da interação.
- Se houver necessidade de customização muito específica, ou o projeto exigir controle total sobre cada detalhe da arquitetura, o LangChain pode ser a melhor escolha.

Conclusão

Não existe um framework “melhor” de forma absoluta: cada um atende a perfis e projetos diferentes. Para a maior parte dos desenvolvedores e para quem está começando, o Agno proporciona ganhos rápidos de produtividade e facilidade no desenvolvimento de agentes de IA. Já o LangChain é ideal para quem busca máxima flexibilidade e está disposto a lidar com uma configuração mais detalhada.

05. Configurando o Agno

Chegou a hora de colocar a mão na massa. Neste capítulo, você vai aprender como configurar seu ambiente local para começar a criar agentes com o Agno, usando o terminal e uma IDE Python. Vamos instalar o gerenciador de pacotes recomendado, criar o ambiente virtual e fazer nossa primeira chamada a um modelo de linguagem.

> **Pré-requisitos:** conhecimentos básicos de Python e uma IDE instalada (como VS Code, Cursor ou PyCharm).

Por que usar o Agno com o Yuvi?

O Agno recomenda o uso do **Yuvi** (uv), um gerenciador de pacotes moderno para Python, por ser muito mais rápido e eficiente que alternativas como `pip` ou `poetry`. O Yuvi facilita a criação de ambientes isolados para cada projeto e agiliza a instalação de bibliotecas.

Instalação do Yuvi

No terminal da pasta do projeto, execute:

```
pip install uv
```

Após isso, inicialize o projeto com:

```
uv init
```

Esse comando cria os arquivos essenciais:

- `.gitignore`
- `pyproject.toml`
- `README.md`
- Versão do Python utilizada

Criando o ambiente virtual

Crie o ambiente virtual com:

```
uv venv
```

Ele será criado em uma pasta própria (`.venv`) e manterá todas as dependências do projeto isoladas.

Instalando o Agno

Com o ambiente pronto, instale o framework Agno:

```
uv add agno
```

Criando o primeiro script

Vamos testar nossa primeira integração com um modelo de linguagem da Groq. A **Groq** é uma empresa de tecnologia - uma startup para ser mais preciso, especializada no desenvolvimento de **hardware e software otimizados para inteligência artificial (IA) e machine learning (ML)**. A principal proposta da Groq é oferecer **aceleração extrema no processamento de modelos de IA**, com foco em **baixa latência**, **altíssimo throughput** e **previsibilidade de desempenho**. Vamos usá-la em primeira instância devido a gratuidade.

Crie um arquivo chamado `0_llm_call.py` e insira o seguinte código:

```
from agno.models.groq import Groq
from agno.models.message import Message

model = Groq(id="llama-3.3-70b-versatile")

message = Message(
    role="user",
    content=[{"type": "text", "text": "Olá, meu nome é Rodrigo"}]
)

response = model.invoke([message])
print(response.choices[0].message.content)
```

Para executar esse script também será necessário ter instalado em sua máquina a biblioteca Groq que pode ser instalada com o seguinte comando:

```
uv add groq
```

Conectando-se à Groq

A **Groq** é uma plataforma que hospeda modelos open-source (como Llama e DeepSeek) com desempenho altíssimo — e o melhor: com acesso gratuito.

Etapas para usar:

1. Acesse console.groq.com e crie uma conta.
2. Gere uma **API key** (ex: `curso-asimov`) e copie o valor.
3. Crie um arquivo `.env` na raiz do projeto com o seguinte conteúdo:

```
GROQ_API_KEY=sua_chave_aqui
```

Carregando variáveis do `.env`

Instale o pacote de suporte com:

```
uv add python-dotenv
```

E adicione ao seu script:

```
from dotenv import load_dotenv
load_dotenv()
```

Executando o código

Você pode rodar o script com:

```
python 0_llm_call.py
```

Ou usar o terminal interativo da sua IDE (como o **Cursor**) para ver os resultados diretamente. Se aparecer um erro solicitando o `ipykernel`, basta instala-lo com o comandouv `pip install ipykernel`.

Entendendo a resposta

O Agno retorna as respostas dos modelos com uma estrutura semelhante à da OpenAI. Se você seguiu o passo a passo corretamente, receberá uma resposta parecida com essa:

Figura 2: Exemplo de saída do nosso primeiro algorítmo

Conclusão

Neste capítulo, você deu os primeiros passos essenciais para começar a trabalhar com inteligência artificial utilizando o Agno. Primeiro, instalou o gerenciador de pacotes **Yuvi** e criou um **ambiente virtual** para manter seu projeto organizado. Em seguida, instalou o **Agno**, configurou uma **chave de acesso da Groq** e realizou sua **primeira chamada a um modelo de linguagem (LLM)**. A partir daqui, começaremos a utilizar a classe `Agent`, que torna o processo de criação de agentes ainda mais simples e eficiente. Se você chegou até aqui, saiba que o mais difícil já passou — agora é hora de explorar todo o potencial do Agno com mais fluidez e autonomia.

06. Criando nosso primeiro Agente

No capítulo anterior, aprendemos como chamar diretamente um modelo de linguagem com o Agno. Apesar de funcionar, esse não é o fluxo recomendado para projetos reais. A partir de agora, vamos trabalhar com a classe `Agent`, que encapsula toda a lógica necessária para interação com LLMs, uso de ferramentas, execução de funções e construção de fluxos de conversação.

Por que usar a classe `Agent`

A classe `Agent` do Agno simplifica a criação de agentes inteligentes ao lidar com:

- Execução de funções automaticamente
- Integração com ferramentas externas (como web search)
- Padronização de chamadas a modelos de linguagem
- Definição de instruções, memórias e objetivos

Em resumo, ela permite que você foque no comportamento do agente e não na infraestrutura que o sustenta.

Criando um agente pesquisador com Tavily

Vamos criar um agente simples chamado **Researcher**, que faz pesquisas na web em tempo real.

Instalação da ferramenta Tavily

Tavily é um serviço de pesquisa online que oferece uma API para buscar informações na internet. O Agno já possui uma integração com ela via a ferramenta `TavilyTools`.

Crie uma conta gratuita em https://www.tavily.com, gere uma API Key e adicione ao seu arquivo `.env`:

```
TAVILY_API_KEY=sua_chave_aqui
```

Instale o pacote Python:

```
uv add tavily-python
```

Código do agente

Crie um arquivo chamado `1_1_researcher.py` com o seguinte código:

```
from agno.agent import Agent
from agno.tools.tavily import TavilyTools
from agno.models.groq import Groq

agent = Agent(
    model=Groq(id="llama-3.3-70b-versatile"),
    tools=[TavilyTools()],
    debug_mode=True
)

agent.print_response("Use suas ferramentas para pesquisar a temperatura de hoje em Porto
↪  Alegre")
```

Execute com:

```
uv run 1_1_researcher.py
```

A resposta mostrará o processo do agente pensando, chamando a ferramenta Tavily e retornando a resposta final.

Criando um agente analista financeiro com Yahoo Finance

Vamos agora criar um agente especializado em buscar informações do mercado financeiro.

Código do agente

Crie o arquivo `1_2_analista.py`:

```
from agno.agent import Agent
from agno.models.groq import Groq
from agno.tools.yahoofinance import YahooFinanceTools
from dotenv import load_dotenv

load_dotenv()

agent = Agent(
    model=Groq(id="llama-3.3-70b-versatile"),
    tools=[YahooFinanceTools()],
    instructions="Use tabelas para mostrar a informação final. Não inclua nenhum outro
↪  texto.",
```

```
    debug_mode=True
)

agent.print_response("Qual a cotação atual da ação da Apple?", stream=True)
```

Instale o pacote necessário:

```
uv add yahoo-finance-python
```

Execute com:

```
uv run 1_2_analista.py
```

A resposta deverá incluir uma tabela com a cotação atual da Apple.

Explorando outras possibilidades

A classe `Agent` permite muitas outras configurações:

- `instructions`: define o comportamento esperado do agente
- `tools`: lista de ferramentas disponíveis
- `debug_mode`: exibe os bastidores da execução
- `stream=True`: ativa resposta contínua estilo ChatGPT

Nas próximas aulas, vamos explorar como dar memória ao agente, conectá-lo a múltiplas ferramentas e torná-lo mais autônomo. ## Conclusão

Neste capítulo, você aprendeu:

- A criar agentes com a classe `Agent`
- A usar ferramentas como Tavily e Yahoo Finance
- A configurar instruções e stream de resposta
- A executar agentes via terminal com `uv run`

Com isso, começamos a estruturar agentes realmente funcionais e conectados com o mundo real. No próximo capítulo, daremos novos poderes ao agente com memória e funções customizadas.

07. Criando nossas próprias tools

Após a criação do nosso primeiro agente e a implementação de ferramentas padrão do Agno, a próxima dúvida é: **como podemos criar nossas próprias ferramentas e oferecê-las ao modelo de linguagem?** Nesta aula, vamos aprender a criar essas ferramentas personalizadas.

Introdução à Criação de Ferramentas

O primeiro passo foi criar um novo arquivo chamado `13_onTools`, onde vamos desenvolver nossas ferramentas personalizadas. Vamos começar com uma ferramenta simples para **converter Celsius para Fahrenheit**. Embora já tenhamos um modelo básico de agente que consulta a temperatura em Porto Alegre, vamos adicionar uma funcionalidade que permita ao agente converter valores de temperatura. Vamos usar o agente anterior como base para usarmos nossa nova ferramenta.

Exemplo de Função: Conversão de Celsius para Fahrenheit

Vamos criar a função de conversão de Celsius para Fahrenheit. A fórmula para a conversão é simples:

$$\text{Temperatura em Fahrenheit} = \left(\text{Temperatura em Celsius} \times \frac{9}{5}\right) + 32$$

Agora, vamos implementá-la em Python:

```python
def celsius_to_fh(temperatura_celsius: float):
    return (temperatura_celsius * 9/5) + 32
```

Essa função simples recebe um valor em Celsius e retorna o valor convertido em Fahrenheit. Agora, podemos passar essa função para o modelo de linguagem, mas ainda não está funcionando da maneira que esperamos. Vamos entender por que.

Por que NÃO Funciona Imediatamente?

Mesmo com a função definida, ela não será reconhecida pelo modelo de linguagem imediatamente. Isso ocorre porque o modelo precisa saber **como usar a função**. Quando instanciamos o agente pela primeira vez, o Agno passa um prompt inicial que lista todas as ferramentas disponíveis, com uma breve descrição de como elas funcionam.

No entanto, **essa função de conversão ainda não está documentada**, e o modelo não saberá usá-la corretamente. O Agno, por outro lado, permite integrar a documentação diretamente no código, o que facilita o processo de informar ao modelo como utilizá-la.

Adicionando a Documentação (Docstring)

A documentação de uma função no Python é feita através do **docstring**, um comentário estruturado que descreve o que a função faz, seus parâmetros e o valor de retorno. No nosso caso, a documentação ficaria assim:

```
def celsius_to_fh(temperatura_celsius: float):

    """
    Converte temperatura de Celsius para Fahrenheit.

    Args:
        temperatura_celsius (float): Temperatura em graus Celsius
    Returns:
        float: Temperatura convertida para Fahrenheit
    """
    return (temperatura_celsius * 9/5) + 32
```

Função de Conversão de Temperatura: Explicação Detalhada

1. **`def celsius_to_fahrenheit(temperatura_celsius: float) -> float:`**
    - `def` indica que estamos definindo uma função.
    - `celsius_to_fahrenheit` é o **nome da função**, que descreve sua ação.
    - `temperatura_celsius: float` é o **parâmetro de entrada** da função, com anotação de tipo indicando que esperamos um número decimal (`float`).
    - `-> float` indica o **tipo de retorno** da função, ou seja, a função devolverá um valor decimal.
2. **Docstring** (`" ... "`)
    - Essa parte é a documentação da função.
    - Explica **o que a função faz**, quais são os **argumentos esperados** e o que a função **retorna**.
    - Essa documentação é importante porque o Agno consegue ler o docstring e informar o modelo de linguagem sobre como utilizar a função.
3. **`return (temperatura_celsius * 9/5) + 32`**
    - Essa linha realiza a **conversão matemática**.

- Primeiro multiplica a temperatura em Celsius por **9/5** (ou 1,8).
- Depois soma **32** para completar a conversão para Fahrenheit.
- `return` significa que o resultado desta operação será enviado como saída da função.

Automatizando o Processo com Ferramentas de Desenvolvimento

Se você estiver usando ferramentas como **Cursor** ou uma extensão no **VS Code** para programação com agentes de IA, a geração da docstring pode ser feita automaticamente. No **Cursor**, por exemplo, você pode usar o comando `Ctrl−K` para gerar a documentação diretamente com a frase "generate docstring".

Isso facilita bastante, pois o próprio modelo de linguagem pode gerar a documentação para você, sem precisar escrever manualmente.

Exemplo de Uso no Agno

Agora que a função está documentada, podemos usá-la no agente. O modelo de linguagem, ao perceber que a função foi definida, poderá utilizá-la em um cenário como o seguinte:

- O usuário solicita:
  "**Qual é a temperatura de hoje em Porto Alegre, em Fahrenheit?**"
- O modelo de linguagem percebe que precisa primeiro fazer uma busca pela temperatura atual (em Celsius) e depois realizar a conversão para Fahrenheit.

```
# Exemplo de código para realizar a pesquisa e conversão temperatura_celsius
= agente.pesquisar_temperatura("Porto Alegre") temperatura_fahrenheit
= celsius_to_fahrenheit(temperatura_celsius)
```

O Agno, ao processar a consulta, executa a pesquisa, obtém a temperatura em Celsius e realiza a conversão automaticamente, retornando a resposta ao usuário:

> **A temperatura de hoje em Porto Alegre é 22°C, o que equivale a 71.6°F.**

Possíveis Erros e Melhorias

Se o modelo não retornar o valor esperado, pode ser necessário revisar o modelo utilizado. Modelos pequenos, como o **70B**, podem cometer erros de interpretação e chamadas de função. Para resultados mais precisos, recomenda-se o uso de modelos maiores, como o **GPT-4.1** ou **Cloud 3.5**.

Embora o modelo tenha retornado uma temperatura de **22°C**, isso pode ter sido um erro de interpretação da resposta da pesquisa. O importante é que a **função de conversão** foi chamada corretamente e o resultado final foi satisfatório, embora com alguns detalhes a serem ajustados.

Conclusão

Neste capítulo, aprendemos como criar funções personalizadas para nossos agentes, como documentá-las corretamente para que o modelo de linguagem possa usá-las e como integrar essa funcionalidade ao Agno. Além disso, vimos como a escolha do modelo de linguagem pode afetar a precisão dos resultados. Nos próximos capítulos, continuaremos a aprender a trabalhar com modelos mais robustos para melhorar ainda mais a eficiência do nosso agente.

08. Usando ChatGPT via API

Após aprender a trabalhar com modelos locais e outros provedores, vamos agora trocar o nosso modelo para um da **OpenAI** — mais especificamente o **GPT-4.1 Mini** — e conhecer o **Playground do Agno**, que oferece uma interface mais amigável que o terminal.

Conectando-se a um Modelo da OpenAI

O procedimento para se conectar a qualquer provedor de modelos é o mesmo: basta buscar pelo nome do provedor (como `GPT`, `Gemini`, `OpenAI`) ou pela palavra `API`.

Neste exemplo, usaremos a **OpenAI**.

> Os principais provedores, como OpenAI, Anthropic (Claude) e Google (Gemini), são pagos.

Entendendo a Precificação

Os modelos cobram por uso de duas formas:

1. **Custo de entrada (input)** — tokens que enviamos ao modelo.
2. **Custo de saída (output)** — tokens que o modelo gera como resposta.

O custo de saída costuma ser mais alto, pois gerar texto é mais custoso do que apenas processá-lo.

Exemplo (valores de agosto/2025):

- **GPT-4.1 Mini**
  Entrada: US$ 0,40 por 1 milhão de tokens
  Saída: US$ 1,60 por 1 milhão de tokens Para referência:
- **1 milhão de tokens = 600 mil palavras**.
- Considerando uma média de 200 palavras/minuto, isso representa cerca de 3 mil minutos (50 horas) de fala.

Conclusão: Hoje, usar modelos via API é extremamente barato. Um depósito inicial de **US$ 5** (cerca de R$ 25–30) pode sustentar projetos por bastante tempo. ## Criando a Conta e Obtendo a API Key

1. Crie uma conta na OpenAI.
2. Vá em **Settings → Billing** e adicione saldo.
3. Gere sua **API Key**.
4. No projeto, adicione a chave ao arquivo `.env`:

```
OPENAI_API_KEY="sua_chave_aqui"
```

Alterando o Modelo no Agno

Com a chave configurada, podemos substituir o modelo anterior (ex.: Grok) por um da OpenAI.

```
from agno.models.openai import OpenAIChat
from agno.agent import Agent

agent = Agent(
    model=OpenAIChat(id="gpt-4.1-mini"),
    # demais configurações...
)
```

Se ainda não tiver a biblioteca instalada:

```
uv add openai
```

Explicação Detalhada

1. **Importações**
    - `from agno.models.openai import OpenAIChat`: importa a classe responsável por integrar com modelos da OpenAI.
    - `from agno.agent import Agent`: importa a classe base para criar agentes no Agno.
2. **Instanciando o agente**
    - `model=OpenAIChat(id="gpt-4.1-mini")`: define qual modelo será utilizado.
    - `id="gpt-4.1-mini"`: especifica a versão do modelo desejada.
3. **Execução**
    - Ao rodar o agente, todas as interações serão enviadas para o GPT-4.1 Mini. ## Testando no Projeto

Podemos reaproveitar o projeto anterior e comparar os resultados:

- **Grok**: interpretou incorretamente a temperatura, retornando valores imprecisos.
- **GPT-4.1 Mini**: entendeu corretamente o texto e retornou a temperatura real (ex.: 18 °C) sem alucinações. ## Usando o Playground do Agno

Embora o terminal seja funcional, ele não é tão visual como você pode ver no exemplo a seguir:

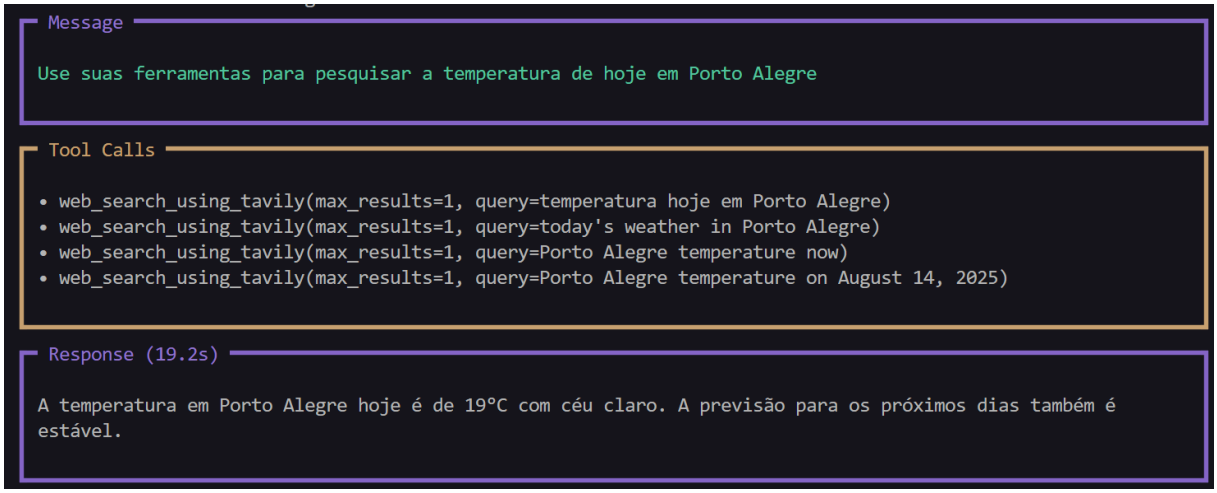

Figura 3: Exemplo no Terminal

O **Playground do Agno** oferece:

- Interface web para execução e depuração de agentes.
- Visualização passo a passo das chamadas e respostas.
- Histórico organizado para referência.

Isso torna o desenvolvimento mais prático e rápido. ## Conclusão

Neste capítulo, aprendemos a:

- Entender a cobrança por tokens.
- Criar conta na OpenAI e gerar a API Key.
- Alterar o código para usar o GPT-4.1 Mini no Agno.
- Comparar resultados entre diferentes modelos.

Com isso, você está pronto para integrar modelos avançados ao seu agente e aproveitar ao máximo os recursos da OpenAI dentro do Agno. No próximo capítulo exploraremos o Playground.

09. Agno Playground

Agora vamos aprender sobre o **Agno Playground**, uma ferramenta muito mais eficiente e visual para o desenvolvimento de agentes. Com o Playground, podemos depurar e testar nossos agentes de uma forma mais fluida e organizada, comparado ao terminal, que é limitado para visualização e interação.

Introdução ao Agno Playground

O Agno Playground é uma interface que permite desenvolver, testar e depurar agentes de maneira muito mais intuitiva. Ao invés de debugar no terminal, onde é difícil acompanhar o fluxo e o histórico, o Playground oferece visualização clara das sessões e memória dos agentes, facilitando todo o processo de desenvolvimento.

Como Configurar o Agno Playground

Para usar o Agno Playground, o processo é simples. Vamos começar.

1. **Comentar o código antigo:** No início, comentamos a linha de código que estava executando o agente diretamente pelo terminal.
2. **Importações adicionais:** Precisamos importar as classes `playground` e `servePlaygroundApp` do Agno.

```
from agno.playground import Playground, serve_playground_app
```

3. **Criando o Playground:** Vamos criar o playground passando a lista de agentes que queremos testar:

```
app = Playground(agents=[
        agent
]).get_app()
```

4. **Configurando o método de execução:** Usamos o método `serve_playground_app`, que exige o nome do arquivo Python onde estamos executando o agente, garantindo que o Playground esteja ativo.

```
if __name__ == '__main__':
    serve_playground_app('13_onTools:app', reload=True)
```

Instalando Dependências Necessárias

Caso o Agno Playground peça bibliotecas adicionais, instale-as utilizando os seguintes comandos:

```
uv add fastapi
uv add uvcore
```

Após essas instalações, basta executar o código e iniciar o servidor do Playground.

Testando o Agno Playground

Agora que o Playground está configurado, podemos testar nosso agente de forma mais interativa.

1. Abra o terminal e execute o script.
2. Após iniciar o Playground com `app.run()`, o servidor local do Agno estará ativo. Para acessá-lo, abra o navegador de sua preferência, digite o endereço do *localhost* e não se esqueça de adicionar `/v1` ao final do endereço.

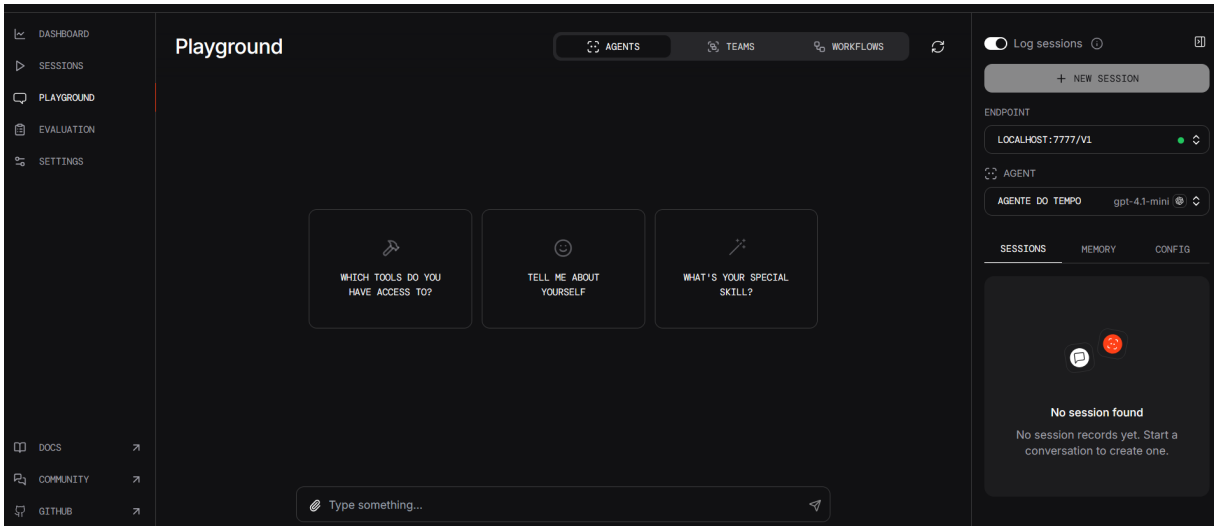

Figura 4: Exemplo do Playground

> **Dica:** Se houver outros agentes rodando na mesma porta, altere a porta ou feche as instâncias anteriores para evitar conflito.

Funcionalidades do Agno Playground

No Agno Playground, temos várias funcionalidades úteis para testar e depurar agentes:

Histórico de Sessões

O Playground registra todas as sessões realizadas, mostrando o histórico completo das interações. Isso facilita a depuração, pois você pode ver exatamente o que aconteceu em cada sessão.

Visualização de Chamadas de Função

Cada vez que o agente utiliza uma função, o Playground exibe como a função foi chamada, com parâmetros e resultados. Isso permite que você acompanhe o fluxo de execução do agente de forma clara.

Documentação e Customização

No Agno Playground, também podemos personalizar o nome dos agentes, o que facilita a organização e documentação do que cada agente faz. No exemplo abaixo, renomeamos o agente para **agente do tempo**, que lida com informações de temperatura.

```
agent = Agent(
    name="Agente do tempo", #nova linha adicionando um nome ao agente
    model=OpenAIChat(id="gpt-4.1-mini"),
    tools=[
        TavilyTools(),
        celsius_to_fh,
        ]
)
```

Teste de Funções no Agente

O Agno Playground permite que você interaja diretamente com o agente no navegador. Por exemplo, se pedirmos para o agente pesquisar a temperatura atual em Pernambuco e para converter de Celsius para Fahrenheit, ele irá utilizar o Tavily e a função de conversão que já definimos para fornecer a resposta correta.

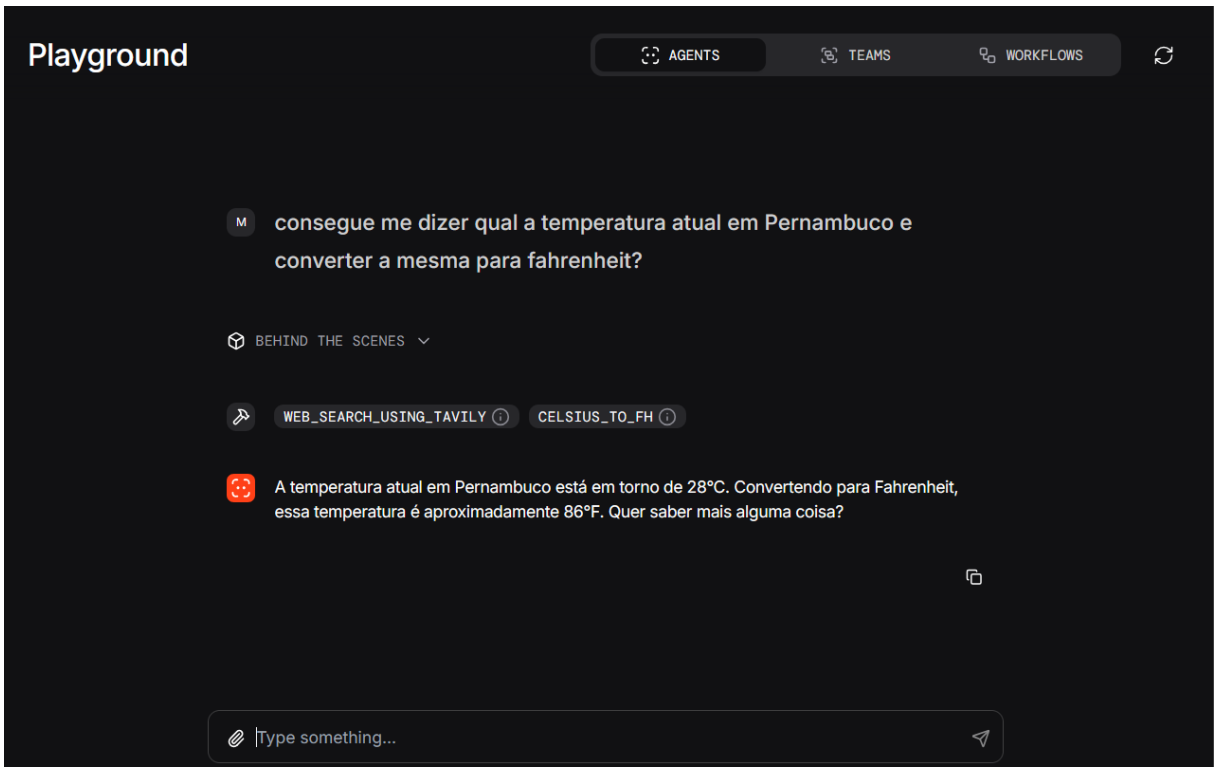

Figura 5: Exemplo de uso das ferramentas

Como pudemos ver, além de exibir a saída, o Playground também nos permite visualizar quais ferramentas o agente utilizou. Nesse caso, ele utilizou as duas ferramentas às quais nós concedemos acesso.

Histórico Detalhado e Sessões de Agentes

Uma das grandes vantagens do Agno Playground é o acesso ao histórico detalhado das interações. Você pode visualizar os detalhes de cada sessão, incluindo o tempo gasto, tokens usados e chamadas de função realizadas.

- **Tokens Gasto:** O Playground mostra o número de tokens utilizados em cada sessão, permitindo estimar os custos de execução.
- **Tempo de Execução:** O tempo que o agente leva para processar as funções também é registrado.

Problemas Comuns

Ao utilizar o **Agno Playground**, é possível se deparar com alguns problemas frequentes. Abaixo listamos os mais comuns e suas soluções:

1. **Navegadores com políticas restritivas**
    - Navegadores como **Brave** e **Safari** podem bloquear algumas funcionalidades do Playground devido às políticas de privacidade e proteção de dados.
    - **Solução:** Recomenda-se usar o **Chrome**, **Edge**, **Firefox**, dentre outros.
2. **Uso do endpoint `v1` no localhost
    - É importante adicionar `/v1` ao final do endereço do **localhost** ao acessar o Playground.
    - Isso garante que o navegador acesse o endpoint correto do servidor, evitando problemas de roteamento ou falta de resposta.
3. - **Problemas com sessões não salvas**
    - Em algumas máquinas, o Agno pode apresentar comportamento inesperado, não salvando corretamente as sessões dos agentes.
    - Situação já foi reportada à equipe do Agno.
    - **Solução provisória:** utilizar a **versão 1.5.0** do Agno Playground, que não apresenta este problema.

Conclusão

Neste capítulo, aprendemos como configurar e utilizar o **Agno Playground** para desenvolver e depurar agentes de forma interativa e eficiente. As principais vantagens incluem:

- Visualização do histórico de sessões e chamadas de função;
- Capacidade de personalizar agentes e atualizar em tempo real;
- Acompanhamento detalhado de tokens gastos e tempo de execução.

Com essas ferramentas, o desenvolvimento de agentes no Agno se torna muito mais fluido e produtivo. No próximo capítulo, exploraremos como trabalhar com memória e sessões em mais detalhes.

10. Memória e Storage no Agno

Agora vamos aprender como implementar memória nos nossos agentes para que eles sejam capazes de manter o contexto de conversas anteriores, e como utilizar o módulo **Storage** do Agno para salvar permanentemente o histórico de sessões.

Introdução à Memória Temporária

O **Agno Playground** é o nosso ambiente padrão para desenvolver e testar agentes, oferecendo muito mais recursos e clareza na visualização das interações do que o terminal. No entanto, por padrão, o agente não mantém o histórico completo das conversas. Isso significa que, se enviarmos uma mensagem informando nosso nome e em seguida enviarmos uma pergunta perguntando o nosso nome, o agente inicialmente não saberá responder corretamente, pois apenas processa as mensagens da interação atual.

Parâmetros `addHistoryToMessages` e `numHistoryRuns`

Para habilitar a memória temporária, podemos usar o parâmetro:

```
addHistoryToMessages=True
```

Esse parâmetro adiciona o histórico das últimas mensagens ao contexto enviado para o modelo. Ele funciona junto com:

```
numHistoryRuns=3
```

O valor **3** indica que apenas as três últimas interações serão enviadas ao modelo. Podemos aumentar esse número conforme a necessidade.

Exemplo de Uso

```
agent = Agent(
    model=OpenAIChat(id="gpt-4.1-mini"),
    tools=[...],
    addHistoryToMessages=True,
    numHistoryRuns=3
)
```

Com `addHistoryToMessages=True`, o agente passa a lembrar informações recentes. Aumentar `numHistoryRuns` expande o contexto, mas também eleva o custo de **tokens**.

> **Dica:** Usar um valor pequeno reduz custos, pois menos histórico é enviado a cada requisição.

Limitações da Memória Temporária

Essa memória é volátil. Se o servidor ou o Playground for reiniciado, o histórico se perde. O histórico exibido no Playground é apenas visual e não é persistido pelo modelo.

Para manter as conversas de forma permanente, precisamos utilizar um **Storage**.

Armazenamento Persistente com `Storage`

O módulo **Storage** do Agno permite salvar conversas e dados do agente no computador ou em bancos de dados externos, garantindo que o histórico não seja perdido entre sessões.

Exemplo com `SQLiteStorage`

O **SQLite** é um banco de dados leve, baseado em arquivo físico, ideal para testes e desenvolvimento.

```
from agno.storage.sqlite import SQLiteStorage

# Instancia o storage SQLite
db = SQLiteStorage(
    table_name="agent_sessions",
    db_file="/temp/agent.db"
)

# Passa o storage para o agente
agent = Agent(
    model=OpenAIChat(id="gpt-4.1-mini"),
    tools=[...],
    storage=db
)
```

- `table_name`: nome da tabela onde as sessões serão armazenadas.
- `db_file`: caminho do arquivo físico que guardará o banco.

> **Atenção:** Se o arquivo for excluído, o histórico será perdido.

Instalando Dependências

O SQLite no Agno utiliza **SQLAlchemy** como backend. Caso não esteja instalado:

```
uv add sqlalchemy
```

Testando o Storage

1. Inicie o agente com o storage configurado.
2. Inicie uma conversa: `Olá, tudo bem com você?`
3. Pare e reinicie o servidor.
4. Ao selecionar novamente o agente no Playground, a conversa anterior estará disponível.

Outras Opções de Storage

Além do SQLite, o Agno suporta:

- **MongoDB** – recomendado para produção.
- **PostgreSQL** – banco relacional robusto.
- **Redis** – armazenamento rápido em memória.
- **YAML** – formato de arquivo simples para persistência local.

Nos próximos projetos, exploraremos outras opções, principalmente para uso em produção.

Conclusão

Neste capítulo vimos como habilitar a memória temporária nos agentes, entendendo a função de `addHistoryToMessages` e `numHistoryRuns` e seus impactos no custo e no contexto de execução. Exploramos também as limitações dessa abordagem e por que, em muitos casos, é necessário configurar um **Storage** para preservar as interações. Conhecemos o uso prático do SQLite para esse fim e refletimos sobre as diferenças entre adotá-lo em ambientes de desenvolvimento e migrar para soluções mais robustas, como PostgreSQL ou MongoDB, quando o projeto evolui para produção. Essa base permitirá que você desenvolva agentes mais inteligentes, capazes de manter e recuperar informações de forma eficiente, garantindo uma experiência contínua para o usuário.

11. Adicionando Conhecimento Externo ao Agente com RAG

Agora que aprendemos a dar ferramentas para o agente, habilitar memória e salvar sessões em banco de dados, vamos avançar para um ponto fundamental: permitir que o agente acesse **conhecimento externo**. Isso é útil quando precisamos trabalhar com grandes volumes de informação, como documentos PDF extensos, repositórios de arquivos ou bases de dados corporativas.

Introdução à Classe `Knowledge`

O Agno fornece a classe `Knowledge`, que permite processar e incorporar diferentes fontes de informação ao modelo. Entre as possibilidades, podemos integrar:

- PDFs locais ou da internet;
- Objetos armazenados em S3 da aws;
- Sites e conteúdos da web (incluindo Wikipedia e YouTube);
- Arquivos JSON e DOCX;
- Papers do **arXiv**;
- Entre outros formatos.

Exemplo Prático com PDF

Para o exemplo, utilizaremos o relatório **Global EV Outlook 2025** sobre o mercado de veículos elétricos. Nosso objetivo é criar um agente capaz de responder perguntas baseadas nesse documento.

Estrutura do Projeto

1. Criar um arquivo `pdfagent.py`.
2. Importar as classes necessárias: `python from agno.agent import Agent from agno.playground import Playground, serve_playground_app from agno.storage.sqlite import SQLiteStorage from agno.models.openai import OpenAIChat from agno.knowledge.pdf import PDFKnowledgeBase, PDFReader from agno.vectordb.chroma import ChromaDB`
3. Instanciar o banco vetorial (**VectorDB**) com `ChromaDB`.
4. Criar a base de conhecimento `PDFKnowledgeBase` apontando para o PDF desejado.
5. Associar essa base de conhecimento ao agente.

O que cada um faz?

- **Agent**: define o agente (modelo + ferramentas + memória/knowledge).
- **Playground / serve_playground_app**: expõe o agente numa UI local.
- **SQLiteStorage**: salva o histórico de sessões em disco.
- **OpenAIChat**: modelo de linguagem usado pelo agente.
- **PDFKnowledgeBase / PDFReader**: cria a base de conhecimento a partir de PDF(s).
- **ChromaDB**: banco vetorial que armazena os embeddings do PDF.

Criando a Base de Conhecimento

```
# Instanciando o banco vetorial
vectordb = ChromaDB(
    collection="pdfAgent",
    persist_directory="/temp/chroma_db"
)

# Criando a base de conhecimento a partir do PDF
knowledge = PDFKnowledgeBase(
    path="GlobalEVOutlook2025.pdf",  # caminho do PDF (pode ser lista ou diretório)
    vectordb=vectordb,               # usa o Chroma definido acima
    reader=PDFReader(chunk=True)     # extrai texto e fatia em pedaços (chunks)
)
```

> **Chunking**: o parâmetro `chunk=True` divide o documento em partes menores para processamento mais eficiente.

Após criar a base, executamos `knowledge.load()` na **primeira vez** para processar e salvar os dados no `VectorDB`.

O Papel do VectorDB e Embeddings

O `VectorDB` transforma textos em **vetores matemáticos** (via embeddings), permitindo buscar informações por **similaridade semântica** e não apenas por correspondência exata de palavras. Isso significa que, se perguntarmos sobre “veículos elétricos”, ele também encontrará trechos que mencionem “carros elétricos”.

Essa abordagem é conhecida como **RAG** (*Retrieval-Augmented Generation*), onde buscamos apenas os trechos relevantes e os passamos ao modelo como contexto para gerar a resposta.

Integrando ao Agente

```
agent = Agent(
    name="Agente de PDF",
    model=OpenAIChat(id="gpt-4.1-mini"),
    storage=db,
    knowledge=knowledge,
    instructions="Você deve chamar o usuário de senhor",
    description="",
    add_history_to_messages=True,
    search_knowledge=True,
    num_history_runs=3,
    # debug_mode=True
)
```

No `__main__`, executamos:

```
if __name__ == '__main__':
    knowledge.load()  # Apenas na primeira execução
    serve_playground_app('pdfagent:app', reload=True)
```

Na primeira execução, vai levar algum tempo, mas após a indexação inicial, o agente estará pronto para responder perguntas sobre o documento.

Testando o Agente

Exemplo de interação:

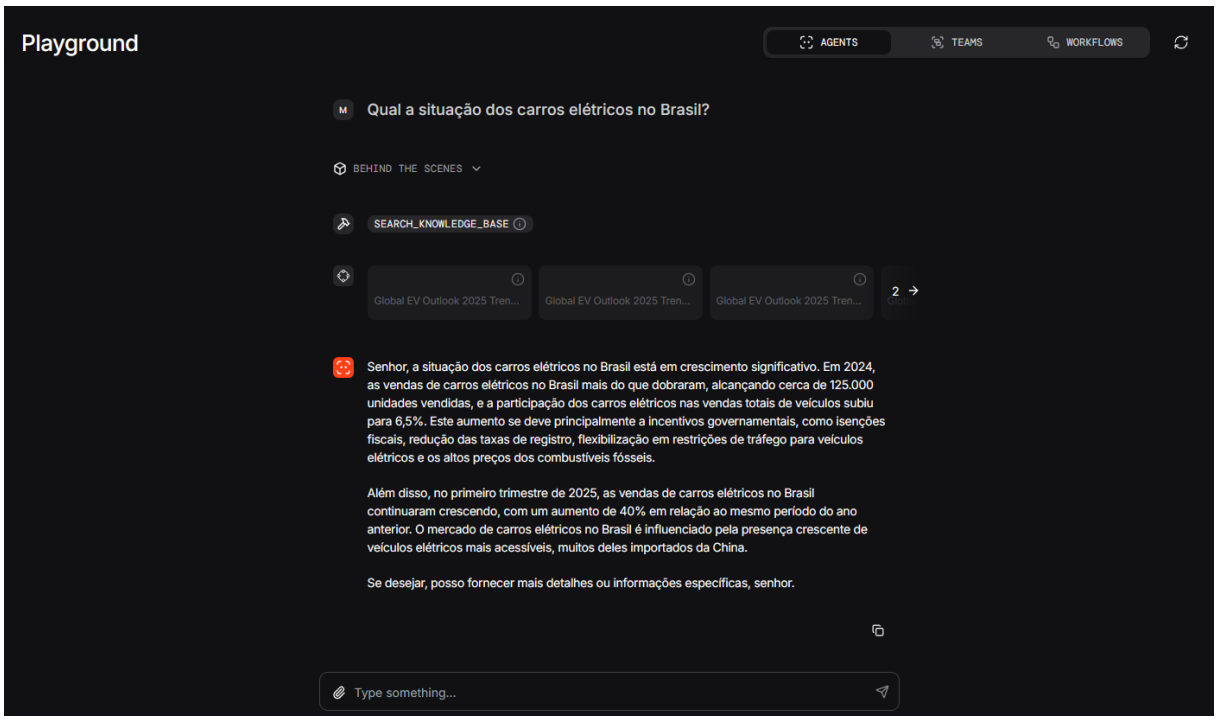

Figura 6: Exemplo com RAG

O agente busca no `VectorDB` os trechos mais relevantes, insere-os no contexto e gera a resposta.

Outras Fontes de Conhecimento

Além de PDFs, o Agno oferece classes para diversas origens:

- **arXivKnowledge** – papers científicos;
- **CSVKnowledge** – arquivos `.csv`;
- **DocxKnowledge** – documentos `.docx`;
- **JSONKnowledge** – arquivos `.json`;
- **WebsiteKnowledge** – captura conteúdo de páginas web e links internos.

A lógica de uso é a mesma: instanciar a base, processar os dados e passá-la como `knowledge` ao agente.

Conclusão

Neste capítulo, exploramos como ampliar o alcance do agente utilizando RAG para integrar conhecimento externo. Vimos como criar uma base de conhecimento a partir de PDFs, armazená-la em um

banco vetorial e habilitar buscas semânticas para fornecer respostas contextualizadas. Esse recurso transforma o agente em um especialista sob demanda, capaz de responder com precisão sobre qualquer acervo de informações que disponibilizarmos, seja um relatório corporativo, uma base científica ou até mesmo um conjunto de sites relevantes.

12. Trabalhando com Memória no Agno

Agora que já dominamos ferramentas, Knowledge/RAG e Storage, vamos explorar a **Memória** do Agno — um mecanismo que registra, de forma **resumida e persistente**, informações relevantes sobre o usuário e a conversa (ex.: nome, preferências, fatos importantes). Diferente do histórico (janela de contexto) e do storage (log completo das sessões), a memória traz **só o essencial** para próximas interações, reduzindo custo de tokens e aumentando personalização.

Diferença entre Memória e Storage

No Agno:

- **Histórico de Mensagens**: contexto imediato enviado ao modelo (controlado por `addHistoryToMessages` e `numHistoryRuns`).
- **Storage**: persistência completa das sessões e histórico em banco de dados.
- **Memória**: registro resumido e estruturado de informações relevantes sobre o usuário ou a interação, gerenciado automaticamente por um "agente auxiliar".

Estrutura Básica da Memória

A memória funciona como um assistente paralelo que:

1. Analisa as mensagens trocadas.
2. Extrai dados importantes (nome, preferências, informações pessoais relevantes ao contexto).
3. Salva esses dados em um banco local (SQLite ou outros).
4. Disponibiliza-os ao agente principal em todas as conversas.

Exemplo Prático

```
from agno.agent import Agent
from agno.tools.tavily import TavilyTools
from agno.models.openai import OpenAIChat
from agno.playground import Playground, serve_playground_app
from agno.memory.v2.memory import Memory
from agno.memory.v2.db.sqlite import SqliteMemoryDb

memory = Memory(
    model=OpenAIChat(id="gpt-4.1-mini"),
    db=SqliteMemoryDb(table_name="user_memories", db_file="tmp/agent.db"),
)
```

```
agent = Agent(
    model=OpenAIChat(id="gpt-4.1-mini"),
    tools=[TavilyTools()],
    instructions="Você é um pesquisador. Responda sempre chamando o usuário de senhor.",
    memory=memory,
    enable_agentic_memory=True,
    # debug_mode=True
)

app = Playground(agents=[
        agent
]).get_app()

if __name__ == "__main__":
    serve_playground_app("31_memory:app", reload=True)
```

Entendendo o código

Configurando a memória (modelo + banco)

```
memory = Memory(
    model=OpenAIChat(id="gpt-4.1-mini"),
    db=SqliteMemoryDb(table_name="user_memories", db_file="tmp/agent.db"),
)
```

- `model=OpenAIChat(...)`: a **memória tem seu próprio LLM**, usado para:
  - detectar trechos “memoráveis” na conversa (ex.: “meu nome é X”),
  - **resumir/estruturar** e gravar no banco (tópico, valor, etc.).
- `db=SqliteMemoryDb(...)`: a memória é **persistida** em `tmp/agent.db`.
  - `table_name="user_memories"`: tabela onde os registros ficam.

Construindo o agente

```
agent = Agent(
    model=OpenAIChat(id="gpt-4.1-mini"),
    tools=[TavilyTools()],
    instructions="Você é um pesquisador. Responda sempre chamando o usuário de senhor.",
    memory=memory,
    enable_agentic_memory=True,
    # debug_mode=True
)
```

- `model`: LLM primário do agente (responde ao usuário e decide usar tools).
- `tools=[TavilyTools()]`: dá **capacidade de pesquisa** ao agente.
- `instructions`: **system prompt**; define tom e conduta ("chame o usuário de 'senhor'").
- `memory=memory`: conecta o **módulo de memórias** criado acima.
- `enable_agentic_memory=True`: habilita o fluxo **agente-com-memória**:
  - o agente pode disparar a **função interna** (ex.: `updateUserMemory`) para registrar fatos importantes automaticamente.
- `# debug_mode=True`: opcional; loga detalhadamente as chamadas de tool/memória.

O restante do código diz respeito a importações e a execução do Playground que nós já conhecemos.

Como Funciona na Prática

1. O usuário diz: *"Meu nome é Rodrigo"*.
2. O agente registra na memória: *Nome do usuário: Rodrigo*.
3. Em qualquer conversa futura, mesmo sem histórico recente, o agente poderá responder: *"Senhor Rodrigo"*.
4. É possível armazenar múltiplos tipos de informações: preferências, dados familiares, histórico de atendimento.

Possibilidades

- Atendimento ao cliente com registro de histórico personalizado.
- Assistentes pessoais que lembram preferências e dados do usuário.
- Agentes que priorizam informações essenciais extraídas de longas conversas.

Conclusão

O recurso de Memória do Agno é uma ferramenta poderosa para criar agentes mais contextuais e personalizados. Ao registrar informações resumidas e relevantes, evitamos depender apenas de janelas de contexto extensas e reduzimos custos com tokens, mantendo a experiência do usuário fluida e contínua. Essa abordagem abre caminho para soluções mais inteligentes em atendimento, suporte e personalização de serviços.

13. Sistemas Multiagentes (Teams) no Agno

Agora que já dominamos ferramentas, RAG/Knowledge e Storage, vamos explorar os **Times (Teams)**: uma forma de combinar **vários agentes especializados** para resolver tarefas complexas. Ao invés de um único agente com muitas ferramentas, dividimos responsabilidades em **membros** com papéis claros e, opcionalmente, um **líder** que roteia, coordena ou sintetiza respostas.

Diferença entre Time e Agente Único

No Agno, você pode resolver problemas de duas formas:

- **Agente Único**: um LLM com um prompt claro e um conjunto de ferramentas. É simples, previsível e suficiente quando o domínio é focado.
- **Time (Team)**: vários agentes com **instruções e ferramentas específicas**, mais um **líder** que organiza o fluxo.

Quando usar **Time**:

- Muitas capacidades heterogêneas (pesquisa web, cálculo, banco de dados, visão, etc.).
- Necessidade de **paralelismo** ou **separação de responsabilidades** (ex.: pesquisador vs. analista).
- Você quer tornar **explícita** a decisão de quem deve atuar em cada etapa.

Quando preferir **Agente Único**:

- Problema estreito, poucas ferramentas.
- Latência/custo e complexidade de coordenação **não compensam**.

Estrutura Básica de um Time

Um **Team** funciona como um orquestrador:

1. **Líder** (opcional, mas recomendado): decide **quem** faz o quê (roteia/coordena/sintetiza).
2. **Membros**: agentes especializados em partes do problema.
3. **Modo de trabalho**: `route`, `coordinate` ou `collaborate`.
4. **Ferramentas por agente**: cada membro vê **apenas** o que precisa.

Exemplo Prático

Vamos construir um time no modo **Route** para **rotear por idioma**. O líder detecta o idioma da pergunta e encaminha ao membro correto (inglês, francês ou chinês). Caso o idioma não seja suportado, usamos **fallback** para inglês.

```
from agno.agent import Agent
# from agno.models.deepseek import DeepSeek
# from agno.models.mistral.mistral import MistralChat
from agno.models.openai import OpenAIChat
from agno.team.team import Team

english_agent = Agent(
    name="English Agent",
    role="You only answer in English",
    model=OpenAIChat(id="gpt-4o"),
)

french_agent = Agent(
    name="French Agent",
    role="You can only answer in French",
    model=OpenAIChat(id="gpt-4o"),
)

multi_language_team = Team(
    name="Multi Language Team",
    mode="route",
    model=OpenAIChat("gpt-4o"),
    members=[english_agent, french_agent],
    show_tool_calls=True,
    markdown=True,
    description="You are a language router that directs questions to the appropriate language
↪  agent.",
    instructions=[
        "Identify the language of the user's question and direct it to the appropriate
↪  language agent.",

        "If the user asks in a language whose agent is not a team member, respond in English
↪  with:",

        "'I can only answer in the following languages: English, French. Please ask your
↪  question in one of these languages.'",

        "Always check the language of the user's input before routing to an agent.",

        "For unsupported languages like Italian, respond in English with the above message.",

    ],
    show_members_responses=True,
)

if __name__ == "__main__":
    # Ask "How are you?" in all supported languages
    multi_language_team.print_response("Comment allez-vous?", stream=True)  # French
    multi_language_team.print_response("How are you?", stream=True)  # English
    multi_language_team.print_response("Come stai?", stream=True)  # Italian
```

Entendendo o código

- **Membros especializados** (`agent_en`, `agent_fr`, `agent_zh`): cada um com **instruções** que fixam o idioma de saída.
- **Líder** (`router`): recebe a pergunta, decide o **destino** e aplica **fallback** quando necessário.
- `**Team(...)**`: instancia o time, define `mode="route"` e permite exibir respostas dos membros com `show_member_responses=True` durante testes.

Como Funciona na Prática

1. O usuário envia uma pergunta em um idioma suportado.
2. O **líder** detecta o idioma e **encaminha** ao membro correspondente.
3. O membro gera a resposta já no idioma correto.
4. Se o idioma não for suportado, o líder responde em **inglês** pedindo que o usuário use um idioma suportado (comportamento de **fallback** que você pode personalizar).

Variações de Modo de Trabalho

Além de `route`, você pode adotar:

- `**coordinate**`** (Coordenação)**: o líder** quebra a tarefa** em subtarefas, **solicita** respostas aos membros e depois **sintetiza** um resultado único.

```
pesquisador = Agent(...)
analista = Agent(...)
coordenador = Agent(...)

team = Team(
    model=OpenAIChat(id="gpt-4o-mini"),
    members=[pesquisador, analista],
    leader=coordenador,
    mode="coordinate",
)
print(team.run("Qual o impacto de juros altos no valuation?"))
```

- `**collaborate**`** (Colaboração)**: todos os membros trabalham** em paralelo** sobre a mesma entrada (ex.: ideação, avaliação e planejamento) e o líder **sintetiza**.

```
ideacao = Agent(...)
avaliacao = Agent(...)
planejamento = Agent(...)
lider = Agent(...)

team = Team(
```

```
    model=OpenAIChat(id="gpt-4o-mini"),
    members=[ideacao, avaliacao, planejamento],
    leader=lider,
    mode="collaborate",
)
print(team.run("Como lançar um curso de data science em 90 dias?"))
```

Possibilidades

- **Assistentes compostos**: pesquisador (web) + analista (cálculo) + redator (síntese).
- **Roteamento por competência**: times por idioma, por domínio (jurídico, financeiro) ou por tipo de entrada (texto, imagem, planilha).
- **Pipelines de decisão**: detecção → análise → verificação → relatório.
- **Paralelismo criativo**: geração de ideias em paralelo seguida de votação/síntese.

Conclusão

Times no Agno são úteis quando há **especializações claras** e a coordenação agrega valor real. Comece simples (um agente único bem instruído) e evolua para **Teams** quando surgirem limitações de roteamento, manutenção de prompts ou necessidade de paralelismo. Defina papéis com clareza, delimite ferramentas por agente e use logs/observabilidade para depurar a colaboração.