

---

## **Criando Multi Agent Systems com CrewAI**

Asimov Academy

# ASIMOV

## Conteúdo

<b>01. Apresentação do curso</b>	<b>4</b>
<b>02. O conceito central de CrewAI</b>	<b>5</b>
<b>03. Entendendo Crews, Agents, Tasks, Tools e Processes</b>	<b>6</b>
Agents . . . . .	6
Tasks . . . . .	6
Crew . . . . .	7
Process . . . . .	7
Tools . . . . .	7
Conclusão . . . . .	8
<b>04. Criação de Agents, Tasks e Crews no CrewAI</b>	<b>9</b>
Overview da aplicação . . . . .	10
Configuração do Ambiente . . . . .	12
Criação dos Agents . . . . .	12
Definição das Tasks . . . . .	13
Criação da Crew . . . . .	14
Ordem importa . . . . .	15
Execução da Crew . . . . .	15
Passando Inputs para a Crew . . . . .	15
Conclusão e Próximos Passos . . . . .	16
Dúvidas sobre criação de Agents, Tasks e Crews . . . . .	17
Desafio - Criação de uma Crew para roteirização de vídeo no YouTube . . . . .	18
<b>05. Princípios da Engenharia de Prompt na criação de Crews</b>	<b>19</b>
Gerando especificidade através da modularização . . . . .	19
Dando Tempo para Pensar Através da Reflexão . . . . .	20
Como Saber se a Minha Task é de Reflexão ou Execução? . . . . .	20
Conclusão . . . . .	21
<b>06. Estruturado Agents e Tasks com ajuda do ChatGPT</b>	<b>22</b>
Prompt 1: Descrevendo o processo . . . . .	22
Prompt 2: Utilizando o Chat para criar Agents e Tasks . . . . .	23
<b>07. Projeto finalizado - Criando uma crew para criar crews</b>	<b>27</b>
agents.yaml . . . . .	27
tasks.yaml . . . . .	28

<b>08. Projeto - Criando uma crew para criar crews</b>	<b>32</b>
Desenvolvendo a primeira versão do sistema . . . . .	32
Criando o arquivo agents.yaml . . . . .	36
Criando o arquivo tasks.yaml . . . . .	37
Refinando as expected_output's . . . . .	38
<b>09. Projeto - Refinando a crew para criar crews</b>	<b>42</b>
Iniciando um projeto . . . . .	42
Analisando a estrutura do projeto . . . . .	43
Arquivo .env . . . . .	44
Arquivo agents.yaml . . . . .	45
Arquivo tasks.yaml . . . . .	45
Arquivo crew.py . . . . .	46
Arquivo main.py . . . . .	48
Conclusão . . . . .	48
<b>10. Observabilidade com AgentOps</b>	<b>49</b>
O que é Observabilidade? . . . . .	49
Introduzindo o AgentOps . . . . .	49
Principais Funcionalidades do AgentOps . . . . .	49
Como Utilizar o AgentOps . . . . .	50
O painel do AgentOps . . . . .	52
Análise de Sessões . . . . .	52
Conclusão . . . . .	53
<b>11. Tools de CrewAI</b>	<b>54</b>
Como um modelo age? . . . . .	54
O que são as tools? . . . . .	54
Os dois principais tipos de tools . . . . .	54
Tools de captura de informação . . . . .	54
Tools de atuação . . . . .	55
Como usar tools? . . . . .	55
Tools disponibilizadas pela CrewAI . . . . .	58
<b>12. Adicionando tools ao meu projeto</b>	<b>62</b>
tasks.yaml . . . . .	63
crew.py . . . . .	64

<b>13. Criando tools presonalizadas para CrewAI</b>	<b>67</b>
Criando e utilizando Tools em CrewAI . . . . .	67
Função que será usada como exemplo . . . . .	67
Criando Tools com BaseTool . . . . .	68
Descrevendo os argumentos . . . . .	69
Juntando tudo . . . . .	70
Utilizando Tools do LangChain . . . . .	70
Integrando Ferramentas do LangChain ao CrewAI . . . . .	71
<b>14. Finalizando o curso</b>	<b>72</b>
Introdução – Recapitulando o curso . . . . .	72
Recapitulação dos conceitos fundamentais . . . . .	72
Aplicações práticas e exemplos . . . . .	73
Dicas e direcionamentos para o futuro . . . . .	73
E pra finalizar . . . . .	73

## 01. Apresentação do curso

Você já parou para pensar que enquanto os agentes autônomos estão apenas começando a mostrar seu potencial, a verdadeira revolução está na criação de sistemas de múltiplos agentes?

Em 2024, assistimos à consolidação da arquitetura de agentes de IA — sistemas autônomos capazes de raciocinar e agir. Entretanto, conforme avançamos, ficou claro que esses sistemas ainda enfrentam limitações. Faltava a eles a especificidade e a capacidade de gerenciar múltiplas ferramentas eficazmente.

É aqui que entram os sistemas multiagentes, uma evolução natural que transforma essa realidade. Imagine o poder de ter vários agentes, cada um especializado em uma determinada tarefa, trabalhando em harmonia e colaboração. Ao invés de depender de um único agente que sabe um pouco de tudo, agora temos uma rede de agentes, todos altamente capacitados em funções específicas. Essa mudança nos permite criar soluções mais robustas e flexíveis, adaptando-se às diversas necessidades de cada usuário.

Para ilustrar, pense em um central de atendimento ao cliente. Na Asimov, recebemos perguntas de todos os tipos: “Qual é a melhor trilha para mim?”, “Como faço o pagamento?”, “Python é a melhor linguagem para IA?”. Com sistemas multiagentes, podemos atribuir cada pergunta a um agente especializado. Assim, se alguém tem uma dúvida sobre a trilha de IA, conversa com o agente que entende essa área; se a dúvida é sobre pagamento, falamos com o agente especializado em vendas. Essa especialização cria um sistema que é não apenas eficiente, mas também extremamente flexível.

Neste curso, vamos aprender a contruir sistemas multiagentes usando a CrewAI, a biblioteca mais renomada nesta área. Abordaremos os conceitos fundamentais necessários para desenvolver qualquer sistema multiagente, explorando tópicos como Agents, Tasks, Crews, Tools e muito mais.

E o melhor: vamos aprender fazendo! Por meio de diversos projetos práticos ao longo do curso, você terá a oportunidade de aplicar o que aprendeu de maneira direta e envolvente.

**Então, seja bem-vindo ao curso: “Criando Multi Agent Systems com CrewAI”! Estamos animados para começar essa jornada ao seu lado e ver as soluções incríveis que você irá criar!**

## 02. O conceito central de CrewAI

Bem-vindos à primeira aula do nosso curso de CrewAI! Hoje, vamos explorar os conceitos essenciais dessa biblioteca, entendendo como ela nos ajuda a construir aplicações inteligentes e colaborativas utilizando sistemas multiagentes.

Antes de começarmos, quero relembrar um ponto fundamental sobre o uso eficiente dos modelos de linguagem:

Nós precisamos induzir os modelos a pensar e agir de maneira similar a humana.

Quando damos a um modelo uma tarefa complexa, por exemplo, “desenvolva um código em Python para um jogo X”, estamos exigindo demais de uma única vez. Um ser humano jamais realizaria esse processo de forma contínua; ele pararia, refletiria sobre onde começar, avaliaria como estruturar o código e quais pontos merecem atenção, para depois desenvolver e, por fim, revisar.

### **Você percebe que uma tarefa complexa exige diversos passos, não?**

Da mesma forma, ao darmos a uma IA generativa a oportunidade de dividir uma atividade em vários passos, obtemos resultados muito superiores do que quando exigimos a execução da tarefa de imediato.

Esse é o conceito central por trás do CrewAI: quebrar tarefas.

Precisamos decompor uma tarefa em pequenos passos e criar agentes específicos para resolver cada um deles. Assim como em uma empresa, onde temos diferentes funcionários com habilidades distintas, um profissional sozinho pode não construir algo de valor, mas, em conjunto com outros profissionais, pode contribuir para a criação de soluções de extrema complexidade.

Mantenha sempre esta ideia em mente a partir de agora! E vamos para o nosso curso.

### 03. Entendendo Crews, Agents, Tasks, Tools e Processes

Nesta aula, vamos apresentar os principais conceitos da biblioteca CrewAI:

- Agents
- Tasks
- Crew
- Process
- Tools

Entendê-los com profundidade é fundamental para utilizarmos bem a biblioteca. Então vamos lá!

#### Agents

Um Agent é uma entidade autônoma com capacidades definidas, capaz de executar diversas funções. Eles são os blocos básicos do CrewAI, responsáveis por tomar decisões e colaborar com outros agents. Ou seja, qualquer estrutura desenvolvida com CrewAI terá, no mínimo, um agent, pois este é o seu elemento fundamental – como o tijolo em uma construção.

#### Qual a característica de um bom agent?

Ele deve ser especializado em uma única área ou função. Se um agent realiza muitas ações diferentes, você não estará aproveitando plenamente o potencial de um sistema multiagente. A ideia é sempre modularizar!

#### Tasks

Depois temos as Tasks, ou tarefas! Ao definir um agent, estabelecemos seu comportamento e seu objetivo principal, mas não determinamos, naquele momento, qual tarefa específica ele executará – isso fica sob responsabilidade das Tasks.

Imagine que um agent defina “quem eu sou”, por exemplo: “sou Adriano, professor de programação Python”. A task, por sua vez, define “o que eu faço” naquele momento, como “desenvolver um curso de CrewAI”.

- Agent: “sou o professor Adriano”
- Task: “desenvolver o curso de CrewAI”

Ou seja: o agent está relacionado à identidade, enquanto a task está atrelada à atividade.

Um agent precisa, no mínimo, ter uma task para agir, mas pode possuir múltiplas tasks, desde que

estas não ultrapassem seu escopo. Pessoalmente, prefiro criar uma única task por agent para garantir maior modularidade na minha crew. Muito raramente eu crio mais de uma task por agent.

E, tanto na definição do agent quanto na da task, a clareza é fundamental para que o sistema funcione bem!

### **Crew**

A estrutura que reúne esses agents e permite que eles interajam é chamada de Crew (ou “tripulação”, em inglês).

É exatamente isso: uma tripulação composta por diversos indivíduos que, trabalhando juntos, direcionam o barco na direção desejada.

Dentro da Crew, são definidos os modos de interação e comunicação entre os agents. Essa arquitetura colaborativa, que discutimos já na aula inicial da nossa trilha de sistemas multiagentes, será detalhada aqui!

### **Process**

E a forma como os agents colaboram será definida pelos Processes, ou processos.

A definição do processo determina como a colaboração entre os agents ocorrerá. Há, basicamente, dois tipos principais:

- No processo sequencial, temos uma sequência pré-estabelecida de tasks a serem realizadas, o que oferece menos autonomia ao sistema, porém maior previsibilidade nos resultados.
- No processo hierárquico, um modelo de linguagem é utilizado para definir a sequência de etapas a ser seguida na execução da Crew. Essa abordagem confere mais autonomia ao sistema, mas reduz o nível de controle e confiabilidade.

Como já discutimos: quanto mais controle, menor autonomia; quanto mais autonomia, menor controle.

### **Tools**

E, não menos importante, temos as Tools do CrewAI.

Na analogia com profissionais de uma empresa, até os melhores colaboradores precisam de ferramentas para facilitar seu trabalho. Eu diria que os melhores profissionais são aqueles que sabem aproveitar ao máximo as ferramentas à sua disposição.



O CrewAI dispõe de diversas tools que ampliam as capacidades dos agents. Elas permitem, por exemplo, acessar bancos de dados, interagir com APIs, processar dados ou realizar operações complexas que seriam inviáveis sem o seu uso.

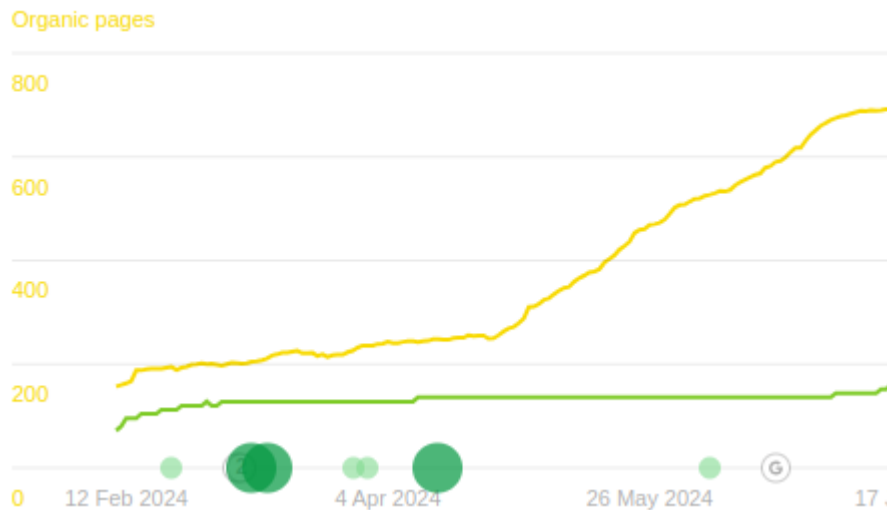
### **Conclusão**

Ao longo desta aula, nós exploramos os componentes essenciais da biblioteca CrewAI: Agents, Tasks, Crew, Process e Tools. Esses elementos formam a espinha dorsal do desenvolvimento de sistemas multiagentes, e compreender como cada um deles funciona é crucial para tirar o máximo de proveito dessa poderosa ferramenta. E na próxima aula, vamos criar nossa primeira crew! Até lá!

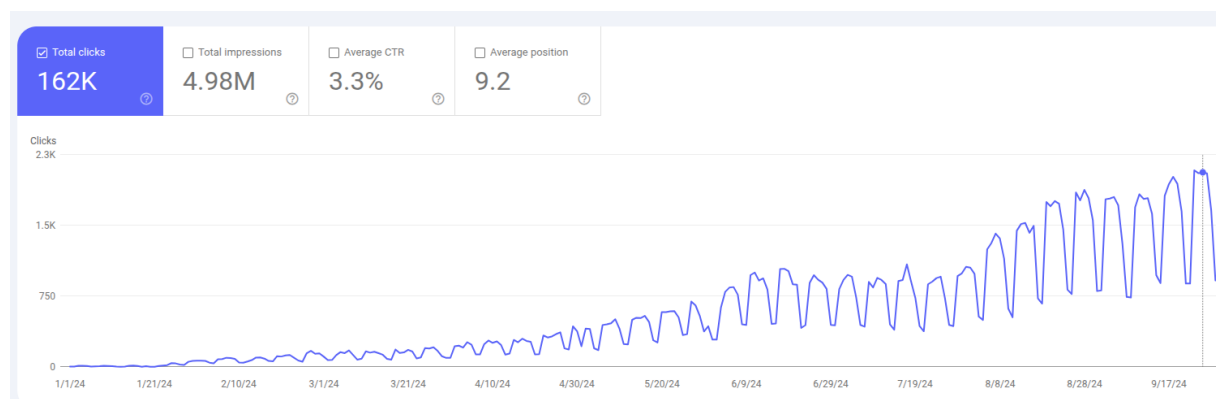
## 04. Criação de Agents, Tasks e Crews no CrewAI

A primeira aplicação de IA que eu criei aqui para a empresa foi um criador de posts de blog. Automatizamos o nosso processo de criação de artigos e, com isso, em menos de 6 meses, geramos mais de 600 posts. O site que recebia 100 acessos orgânicos passou a ter 2 mil.

### Crescimento do número de páginas orgânicas



### Aumento no número de Cliks



A partir desse case, percebemos que precisávamos desenvolver uma trilha de IA aqui na Asimov. O poder dessas LLMs (Large Language Models) combinadas à programação é muito impactante; o ganho de produtividade que tivemos foi incrível e sentimos a necessidade de compartilhar essa experiência com nossos alunos.

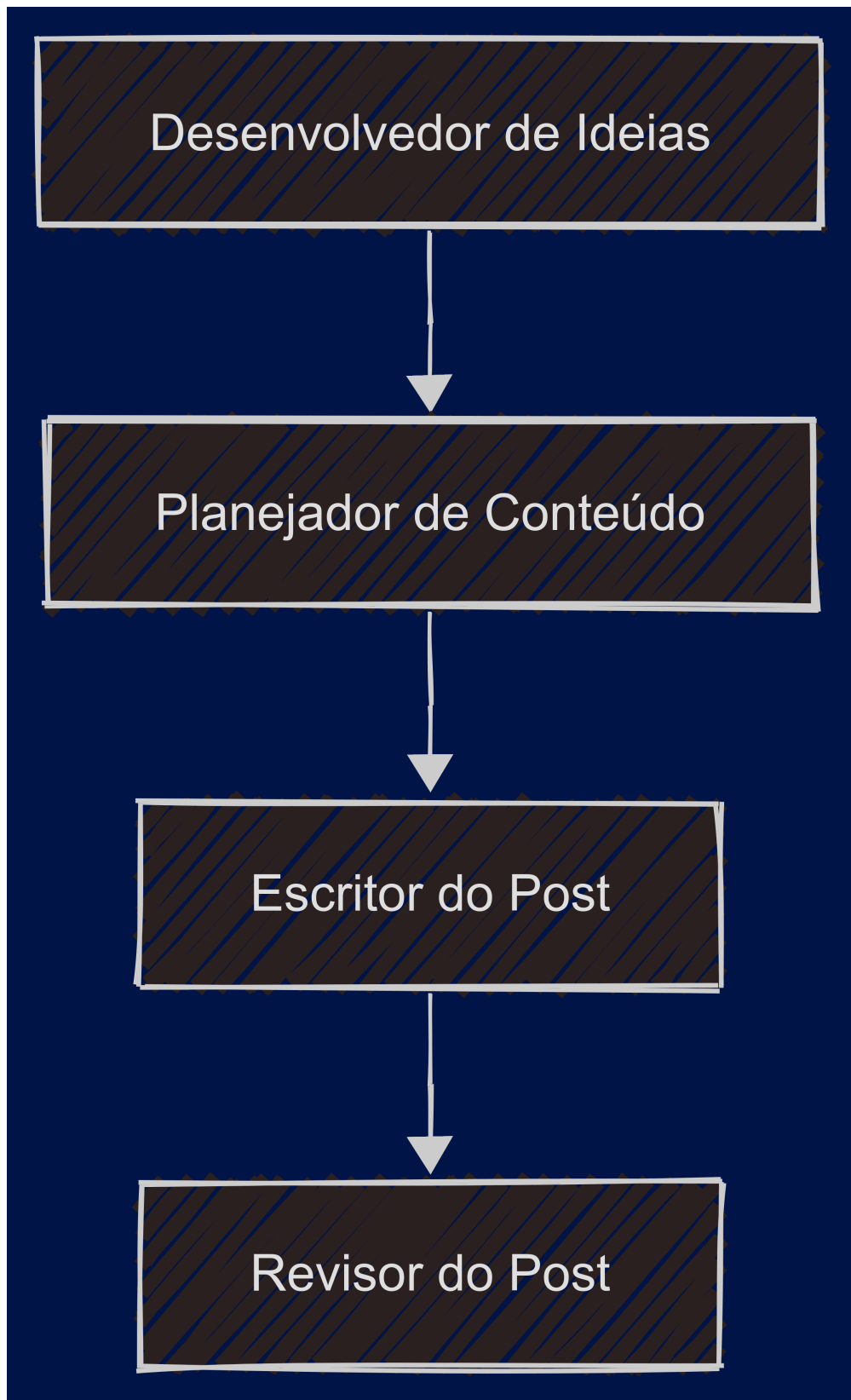
Em homenagem a esta pequena história, decidi criar uma Crew de desenvolvimento de posts de blog como um primeiro exemplo para vocês.

Na aula de hoje, nós finalmente iremos para o código. Aprenderemos como criar Agents, Tasks e uma Crew que os combine, em seu estado mais simples. Apesar de começarmos de forma elementar, a aplicação já será muito interessante e prática.

### Overview da aplicação

Nesta aula, vamos desenvolver uma Crew que consiste em quatro agents, cada um com tasks específicas que serão responsáveis por executar. Teremos os seguintes agents:

- **Desenvolvedor de Ideias:** responsável por gerar ideias criativas.
- **Planejador de Conteúdo:** encarregado de planejar o conteúdo.
- **Escritor do Post:** responsável pela redação do post.
- **Revisor do Post:** que revisa o texto finalizado.



**Figure 1:** Estrutura da Crew

Esses agents terão as seguintes tasks:

- **Desenvolvedor de Ideias**

- **Criar Ideias:** Gera uma lista com 10 ideias criativas e relevantes para posts de blog sobre um tema específico, assegurando a diversidade nas sugestões.
- **Selecionar Ideias:** Analisa a lista de ideias geradas e seleciona a melhor, justificando a escolha com base na relevância e no alinhamento com os objetivos do blog.

- **Planejador de Conteúdo**

- **Planejar Conteúdo:** Cria um briefing detalhado, incluindo informações essenciais como objetivo, público-alvo, tom de voz, palavras-chave e formato do post.

- **Escritor do Post**

- **Escrever Post:** Redige o conteúdo completo do post de blog, seguindo as diretrizes do briefing e a ideia selecionada, garantindo que o texto seja envolvente e bem estruturado.

- **Revisor do Post**

- **Revisar Post:** Revê o post de blog, corrigindo erros gramaticais e de pontuação, e melhorando a fluidez do texto, garantindo que o conteúdo esteja alinhado com o briefing.

Agora, vamos passo a passo criando a aplicação!

## Configuração do Ambiente

Vamos iniciar importando as bibliotecas necessárias, carregando as variáveis de ambiente e importando os elementos essenciais da biblioteca CrewAI:

```
from dotenv import load_dotenv, find_dotenv

_ = load_dotenv(find_dotenv())

from crewai import Crew, Process, Agent, Task
```

**Observação:** A variável de ambiente carrega o arquivo `.env`, que deve estar junto ao projeto. Nele, devemos guardar uma variável chamada `OPENAI_API_KEY` com a chave da API da OpenAI. Desta forma, estaremos executando a Crew através da OpenAI.

## Criação dos Agents

Vamos à criação dos agents. Cada agent é definido com parâmetros essenciais como `role`, `goal`, `backstory` e `verbose`. Esses parâmetros determinam a função do agent no sistema.

O parâmetro `role` descreve a função do agent, enquanto `goal` e `backstory` detalham o objetivo e o contexto do agent. O parâmetro `verbose`, quando definido como `True`, permite que o agent registre informações detalhadas durante sua execução.

Aqui está o trecho de código para a criação dos agents:

```
desenvolvedor_de_ideais = Agent(
    role='Criador de Ideias Criativas para Blog Posts',
    goal='Gerar ideias originais e relevantes para posts de blog dentro de um tema específico',
    backstory='Você é um criador criativo, sempre antenado nas últimas tendências e com uma
    ↪ habilidade impressionante para transformar conceitos em ideias inovadoras. Sua curiosidade
    ↪ e energia o ajudam a criar sugestões únicas que atraem o público-alvo de forma eficaz.',
    verbose=True
)

planejador_de_conteudo = Agent(
    role='Estrategista de Conteúdo para Blogs',
    goal='Planejar e estruturar o conteúdo de maneira eficaz, com base no briefing fornecido',
    backstory='Você é um estrategista detalhista, apaixonado por alinhar objetivos e dados com a
    ↪ criação de conteúdo. Você adora criar planos bem estruturados que orientam os redatores
    ↪ para alcançar os melhores resultados. Seu foco está sempre em garantir que o conteúdo
    ↪ atenda às expectativas do público e aos objetivos de marketing.',
    verbose=True
)

escritor_do_post = Agent(
    role='Redator Criativo de Blog Posts',
    goal='Escrever posts de blog envolventes e de alta qualidade, seguindo o briefing e as
    ↪ diretrizes definidas',
    backstory='Você é um escritor versátil, capaz de adaptar seu estilo de escrita ao tom e ao
    ↪ formato desejado. Seu objetivo é sempre criar conteúdo claro, interessante e que prenda a
    ↪ atenção do leitor, transformando ideias e informações em histórias envolventes e bem
    ↪ estruturadas.',
    verbose=True
)

revisor_do_post = Agent(
    role='Revisor de Conteúdo de Blog Posts',
    goal='Garantir que o post esteja livre de erros e pronto para ser publicado',
    backstory='Você é um revisor minucioso e atento aos detalhes. Sua missão é corrigir erros
    ↪ ortográficos, melhorar a fluidez do texto e garantir que o conteúdo esteja perfeitamente
    ↪ alinhado com o briefing e os padrões de qualidade. Seu trabalho é garantir que cada post
    ↪ esteja impecável antes da publicação.',
    verbose=True
)
```

## Definição das Tasks

Após a criação dos agents, passamos para as tasks, que representam as atividades que cada agent deve executar. Cada task é composta por uma `description` que orienta o que deve ser feito e um `expected_output` que descreve o resultado esperado, ajudando a validar a execução.

No nosso código, definimos várias tasks, cada uma vinculada ao agent especializado para aquela fase do processo de criação do blog. Abaixo está o código que define essas tasks:

```
cria_ideias = Task(
    description='Crie uma lista com 10 ideias diferentes para posts de blog sobre o tema: {tema}.
    ↪ As ideias devem ser criativas, relevantes e diversificadas em formato, com foco em engajar
    ↪ o público-alvo.',
    agent=desenvolvedor_de_ideais,
    expected_output='Uma lista com 10 ideias de posts, cada uma com um título criativo e um breve
    ↪ resumo do que o conteúdo abordará.'
)

seleciona_ideias = Task(
    description='Selecione a melhor ideia da lista gerada, justificando sua escolha com base na
    ↪ relevância e no alinhamento com os objetivos do blog.',
    agent=desenvolvedor_de_ideais,
    expected_output='A escolha de uma ideia com uma justificativa clara e concisa sobre sua
    ↪ relevância e alinhamento com os objetivos do conteúdo.'
)

planeja_conteudo = Task(
    description='Crie um briefing detalhado para o post de blog, incluindo informações como
    ↪ objetivo, público-alvo, tom de voz, palavras-chave e formato.',
    agent=planejador_de_conteudo,
    expected_output='Um briefing estruturado, abordando todos os pontos importantes para guiar a
    ↪ criação do conteúdo.'
)

escreve_post = Task(
    description='Escreva o conteúdo completo do post de blog, seguindo as diretrizes do briefing
    ↪ e a ideia selecionada. Certifique-se de que o post seja envolvente, bem estruturado e
    ↪ adequado ao público-alvo.',
    agent=escritor_do_post,
    expected_output='Um post de blog com introdução, desenvolvimento e conclusão, que seja claro,
    ↪ interessante e alinhado com o briefing.'
)

revisa_post = Task(
    description='Revise o post de blog, corrigindo erros gramaticais e de pontuação, além de
    ↪ melhorar a fluidez do texto. Assegure que o post esteja alinhado com o tom e objetivo
    ↪ definidos no briefing.',
    agent=revisor_do_post,
    expected_output='O post revisado, sem erros gramaticais e com boa fluidez, pronto para ser
    ↪ publicado.'
)
```

## Criação da Crew

Por fim, unimos todos os elementos ao definir a Crew. A Crew é a estrutura que organiza a interação entre os agents e as tasks. Durante a definição da crew, informamos uma lista dos agents envolvidos, outra lista das tasks a serem executadas e especificamos o tipo de processo que a crew seguirá. Neste

exemplo, optamos pelo processo sequencial, o que significa que as tarefas serão executadas em uma ordem pré-definida.

Abaixo, o código que configura a Crew:

```
blog_post_creation_crew = Crew(
    agents=[desenvolvedor_de_ideais, planejador_de_conteudo, escritor_do_post, revisor_do_post],
    tasks=[cria_ideias, seleciona_ideias, planeja_conteudo, escreve_post, revisa_post],
    process=Process.sequential
)
```

### Ordem importa

Muito importante salientar que a ordem em que organizamos nossas tasks é crucial ao desenvolvermos a crew. Quando utilizamos o processo sequencial, a execução das tarefas segue a ordem pre-definida na lista de tasks, com a saída de uma tarefa servindo como contexto para a próxima.

Ou seja, a primeira tarefa será executada, seguida pela segunda, que terá acesso ao resultado da primeira; depois, a terceira, que contará com o resultado da segunda, e assim por diante.

Portanto, é fundamental que você preste atenção ao listar suas tasks de uma maneira que faça sentido dentro do fluxo de trabalho!

### Execução da Crew

E para finalmente executá-la:

```
result = blog_post_creation_crew.kickoff({'tema': 'Aplicações com inteligência artificial'})

print(result.raw)
print(result.tasks_output[0])
```

Nesse exemplo, o método `kickoff()` inicia todo o fluxo de trabalho, coordenando a execução das tasks pelos agentes e retornando um resultado que pode ser analisado tanto de forma geral quanto de forma específica, como a saída da primeira task.

### Passando Inputs para a Crew

Você deve ter percebido que, ao executarmos a crew com o método `kickoff`, passamos o parâmetro `tema` com o valor “Aplicações com inteligência artificial”. Dessa forma, conseguimos direcionar o desenvolvimento do nosso artigo para o conteúdo que gostaríamos de explorar.



**Mas como essa variável tema é utilizada dentro da Crew?** Caso você não tenha observado, em algumas partes dos textos que definimos em nossas Tasks e Agents, deixamos uma variável entre colchetes chamada {tema}.

Veja o exemplo da task atribuída ao **Desenvolvedor de Ideias**:

```
cria_ideias = Task(
    description='Crie uma lista com 10 ideias diferentes para posts de blog sobre o tema: {tema}.
    ↳ As ideias devem ser criativas, relevantes e diversificadas em formato, com foco em engajar
    ↳ o público-alvo.',
    agent=desenvolvedor_de_ideias,
    expected_output='Uma lista com 10 ideias de posts, cada uma com um título criativo e um breve
    ↳ resumo do que o conteúdo abordará.'
)
```

Na execução da crew, essa variável será substituída pelo valor que passamos no método `kickoff`. Este é um dos recursos poderosos do CrewAI: a capacidade de passar inputs dinamicamente para suas tasks e agents. Neste exemplo, utilizamos a variável `tema`, mas poderíamos ter passado múltiplos parâmetros diferentes. A flexibilidade que temos permite que essa variável seja aplicada em diversas partes da nossa aplicação, garantindo que os agents utilizem informações atualizadas e relevantes ao longo do processo.

**Importante!** Essas variáveis do CrewAI podem ser incluídas em qualquer parte dos parâmetros `role`, `goal` e `backstory` dos agents, assim como nos parâmetros `description` e `expected_output` das tasks, proporcionando uma excelente flexibilidade na hora de passar variáveis para a aplicação.

---

## Conclusão e Próximos Passos

É com grande alegria que chegamos ao final desta aula, onde iniciamos nossa jornada com o CrewAI desenvolvendo uma aplicação que reflete a essência de como tudo começou aqui na Asimov em relação às aplicações de IA. Ao criar uma Crew para produção de posts de blog, você não apenas compreendeu a funcionalidade e a importância das classes *Agents*, *Tasks* e *Crew*, mas também deu o primeiro passo em direção a um entendimento mais profundo sobre como otimizar processos com inteligência artificial.

Ao longo deste caminho, percebemos que cada agent tem um papel específico, configurado com parâmetros que orientam sua atuação. As tasks delimitam as atividades que cada agent deve executar e, finalmente, a crew se torna o laço que une esses sistemas, organizando e orquestrando as interações entre os agentes e as tarefas.

Parabéns por ter criado sua primeira crew e por dar esse passo importante na sua trajetória de aprendizado em IA! É emocionante ver como, ao combinar o poder das LLMs e a estrutura do CrewAI, podemos aumentar a produtividade e a eficiência em processos criativos.

### Dúvidas sobre criação de Agents, Tasks e Crews

Caso tenha alguma dúvida sobre como utilizar qualquer uma das classes, criamos a tabela abaixo para ajudá-lo a consultar as funcionalidades de cada parâmetro:

---

Classe	Parâmetro	Descrição
--------	-----------	-----------

---

Agent	role	Descreve a função do agent dentro do sistema, indicando qual tarefa ele está designado a executar.
	goal	Define o objetivo principal do agent, ou seja, o que se espera que ele alcance ao executar sua função.
	backstory	Fornecer um contexto que ajuda a moldar a personalidade e a abordagem do agent em suas tarefas.
	verbose	Permite que o agent registre informações detalhadas durante sua execução quando definido como True, facilitando a depuração e análise de performance.
Task	description	Conta sobre o que a task deve realizar, detalhando as ações específicas que o agent deve executar.
	agent	Indica qual agent é responsável pela execução da task.
	expected_output	Descreve o resultado esperado da task, que é utilizado para validar se a execução foi bem-sucedida.
Crew	agents	Recebe uma lista de agents que farão parte da crew, permitindo a definição da equipe e suas especializações.
	tasks	Recebe uma lista de tasks que serão executadas pela crew, organizando o fluxo de trabalho.
	process	Define como as tasks serão executadas, seja de forma sequencial ou conforme um fluxo hierárquico, estabelecendo a dinâmica da colaboração.

---

## Desafio - Criação de uma Crew para roteirização de vídeo no YouTube

Neste desafio, você irá criar uma Crew utilizando o CrewAI para roteirizar um vídeo para YouTube. O objetivo é automatizar o processo criativo de desenvolvimento de conteúdo de vídeo, dividindo as responsabilidades entre diferentes agentes especializados.

### Agentes a serem Criados:

1. **Pesquisador:** Este agent será responsável por pensar em até 10 ideias diferentes para vídeos em um determinado tema e selecionar a melhor ideia.
2. **Criador da Ideia Central:** Depois que a ideia for escolhida, esse agent deverá formular a principal mensagem que será transmitida durante o vídeo.
3. **Criador do Hook:** Este agent deverá desenvolver os primeiros 30 segundos do vídeo, criando uma abertura atrativa que capture a atenção do público.
4. **Criador do Roteiro:** Este agent será encarregado de desenvolver o roteiro completo do vídeo, estruturando as seções de forma clara e coerente.

### O que você deve fazer:

- Defina os agents com suas respectivas responsabilidades e histórias.
- Crie as tasks que cada agent deve executar.
- Organize todos os agents e tasks em uma Crew e configure o processo de execução.
- Utilize uma variável de input, como `tema`, e passe-a para a Crew usando o método `kickoff()` para ver como essa entrada pode moldar o conteúdo gerado.

Lembre-se de que você pode adicionar outros agents se achar necessário, sempre visando otimizar o processo criativo.

Ao concluir o desafio, sinta-se à vontade para compartilhar sua implementação! Lembre-se de que, no desenvolvimento de aplicações, **o importante não é acertar em tudo**, já que há infinitas formas de se estruturar uma solução, mas sim praticar e experimentar. Cada tentativa irá lhe proporcionar aprendizado e ajudar a aperfeiçoar suas habilidades com o CrewAI. Boa sorte e divirta-se criando!

## 05. Princípios da Engenharia de Prompt na criação de Crews

Ótimo! Agora você já tem um entendimento básico de como criar uma Crew. Já conhece a utilização das três principais classes: *Agent*, *Task* e *Crew*, e pode começar a se desafiar, criando aplicações diferentes.

Mas antes de você começar, quero reforçar alguns pontos que considero essenciais, afinal, sua crew será tão boa quanto sua capacidade de entender o seu problema e dividi-lo em partes.

É isso mesmo! A ideia por trás de uma crew é criar uma aplicação que aumente a capacidade de um modelo de linguagem e, para isso, precisamos usar os dois princípios da engenharia de prompts:

1. Especificidade
2. Dar tempo ao modelo para pensar

Vocês já devem estar familiarizados com esses princípios, mas quero reforçá-los para garantir que vocês criem aplicações úteis!

### Gerando especificidade através da modularização

Para gerar especificidade, você precisa ser capaz de modularizar o seu projeto, pensando no seu problema como uma sequência de etapas que geram um produto final.

Já utilizamos a analogia de um ambiente empresarial, e a retomo aqui porque se aplica perfeitamente. Dentro de uma empresa, temos diversos cargos específicos executando papéis e funções diferentes. Cada cargo necessita de habilidades e ferramentas distintas. Alguns cargos respondem a outros, uns criam, outros revisam, alguns coletam informações e outros levam essa informação ao cliente final. Às vezes, múltiplas funções são executadas por um mesmo funcionário; em outras ocasiões, temos um funcionário responsável por apenas uma única função.

As possibilidades são infinitas, mas algumas regras precisam ser respeitadas para criar uma boa estrutura organizacional, como:

- Tarefas precisam ser específicas.
- Toda tarefa precisa de um responsável.
- Todo o processo precisa de revisão e validação.
- Os canais de comunicação entre os elementos da equipe devem ser claros.

Isso se aplica tanto a um ambiente empresarial quanto a uma Crew!

## **Dando Tempo para Pensar Através da Reflexão**

Na minha opinião, a técnica que mais tem aumentado a capacidade dos modelos de linguagem é a reflexão, e isso é incrível, pois é muito simples.

Já falamos também que devemos pensar no modelo como um ser humano. Quando damos uma tarefa muito complexa para alguém e não damos tempo para que essa pessoa raciocine, a resposta naturalmente será de baixa qualidade. Da mesma forma, acontece com os modelos de linguagem. Para termos bons resultados, é importante incentivá-los a refletir sobre os passos críticos para chegarmos a uma boa resposta.

Mas como fazemos isso no contexto do CrewAI?

Simple, criando agents responsáveis por planejar, pensar e revisar!

Essas são tarefas de reflexão, onde qualificamos nosso trabalho, em contraste com as tarefas de execução, que envolvem o desenvolvimento prático da tarefa. Adicionar passos de reflexão no meio da sua crew é fundamental para obter bons resultados. No exemplo da aula anterior, temos um desenvolvedor de blog posts. Se vocês prestarem atenção nas características das tasks, notarão que temos três tarefas de reflexão e apenas uma de execução:

- Gerar ideias > reflexão
- Planejar conteúdo > reflexão
- Escrever post > execução
- Revisar post > reflexão

Isso foi pensado. Eu poderia ter criado uma crew com apenas um passo, o escrever post, mas o resultado teria sido péssimo, semelhante a pedir para o ChatGPT escrever um post sobre um assunto (o famoso zero-shot prompt). Quando adiciono todas essas etapas de reflexão, a qualidade final melhora muito, e com modelos simples superamos facilmente modelos muito mais complexos.

## **Como Saber se a Minha Task é de Reflexão ou Execução?**

Isso é simples: tasks de reflexão geralmente começam com palavras como planeje, revise, critique, selecione. Ou seja, elas não criam o resultado em si, mas oferecem ferramentas para facilitar a criação ou realizam o polimento necessário para que o resultado atenda às necessidades do usuário.

Já as tarefas de execução realmente desenvolvem o resultado e, em geral, começam com palavras como gere, desenvolva, etc. Podemos ter crews compostas apenas por tarefas de execução (embora o resultado não seja muito bom), mas não podemos ter uma crew composta apenas por tarefas de reflexão.

## **Conclusão**

Quero que vocês saiam desta aula com a clareza de que a principal valência das estruturas multi-agent é a modularização, que permite uma maior especificidade em nossos sistemas.

Além disso, lembrem-se de dar tempo ao modelo para pensar, adicionando etapas de reflexão que farão com que ele planeje, revise e critique as ações, aumentando assim a qualidade da resposta.

Mantendo esses princípios em mente, tenho certeza de que vocês construirão ótimas aplicações!

## 06. Estruturado Agents e Tasks com ajuda do ChatGPT

Muitos processos são simples de modularizar, principalmente aqueles que já conhecemos bem. Agora, quando temos um processo novo, que ainda não pensamos profundamente sobre, sempre podemos utilizar o ChatGPT para auxiliar no processo de modularização.

Vamos ver isto especificamente nesta aula!

### Prompt 1: Descrevendo o processo

Pegando novamente nosso exemplo de escrita de um post de blog, posso solicitar o seguinte:

Dado um processo de criação de um artigo de blog

Quebre o processo em etapas, com cargos e tarefas executadas por esses cargos, para s

Ponto inicial: o tema do conteúdo do post que será definido pelo usuário

Ponto final: texto pronto e revisado

Número máximo de etapas: 5

E resposta que ele me deu é a seguinte:

---

### \*\*Etapa 1: Briefing e Pesquisa\*\*

- \*\*Cargo Principal:\*\* Estrategista de Conteúdo
- \*\*Cargos de Apoio:\*\* Pesquisador / Analista de SEO
- \*\*Tarefas:\*\*
  - \*\*Definição de Objetivos:\*\* Entender o propósito do artigo, público-alvo e os resultados esperados
  - \*\*Levantamento de Informações:\*\* Coletar dados, estatísticas e referências relevantes
  - \*\*Pesquisa de Palavras-Chave:\*\* Identificar termos e tendências para orientar a escrita
  - \*\*Elaboração do Briefing:\*\* Documentar todos os insights e direcionamentos para a equipe

### \*\*Etapa 2: Planejamento e Estruturação do Conteúdo\*\*

- \*\*Cargo Principal:\*\* Editor de Conteúdo
- \*\*Cargos de Apoio:\*\* Redator Sênior / Copywriter

### - \*\*Tarefas:\*\*

- \*\*Criação do Outline (Roteiro):\*\* Definir a estrutura do artigo, com tópicos e sub-tópicos.
- \*\*Definição do Tom e Estilo:\*\* Estabelecer a voz do texto e os elementos que devem ser incluídos.
- \*\*Organização das Ideias:\*\* Distribuir o conteúdo de forma lógica, alinhada com o objetivo do artigo.

### ### \*\*Etapa 3: Redação do Artigo\*\*

#### - \*\*Cargo Principal:\*\* Redator de Conteúdo

#### - \*\*Tarefas:\*\*

- \*\*Produção do Rascunho Inicial:\*\* Escrever o artigo seguindo o outline, integrando as ideias e informações coletadas.
- \*\*Desenvolvimento da Narrativa:\*\* Garantir coerência, fluidez e engajamento, adaptando o tom e o estilo.
- \*\*Incorporação de Chamadas para Ação (CTAs):\*\* Inserir elementos que incentivem o leitor a tomar uma ação.

### ### \*\*Etapa 4: Revisão e Edição\*\*

#### - \*\*Cargo Principal:\*\* Revisor de Texto / Editor

#### - \*\*Tarefas:\*\*

- \*\*Revisão Ortográfica e Gramatical:\*\* Corrigir erros de digitação, gramática e pontuação.
- \*\*Ajustes de Coerência e Coesão:\*\* Verificar a fluidez do texto, melhorar a estrutura e a conexão entre as ideias.
- \*\*Verificação de SEO e Consistência de Estilo:\*\* Confirmar que o texto atende às diretrizes de SEO e mantém um estilo consistente.
- \*\*Feedback e Ajustes:\*\* Se necessário, retornar o conteúdo ao redator para adequações.

### ### \*\*Etapa 5: Aprovação Final e Entrega\*\*

#### - \*\*Cargo Principal:\*\* Gestor de Conteúdo / Editor Final

#### - \*\*Tarefas:\*\*

- \*\*Validação do Conteúdo:\*\* Conferir se todas as etapas anteriores foram cumpridas e se o conteúdo está completo e alinhado.
- \*\*Integração de Feedbacks:\*\* Realizar os últimos ajustes com base nas sugestões recebidas.
- \*\*Preparação para Publicação:\*\* Formatar o texto de acordo com as diretrizes da publicação.

Perfeito! Agora temos uma ótima base para a criação de agents e tasks a partir do que o ChatGPT nos indicou!

## Prompt 2: Utilizando o Chat para criar Agents e Tasks

Além disso, podemos ir além e utilizar o chat para criar os argumentos necessários para a criação dos nossos agents e tasks. Se vocês lembrarem, temos os seguintes parâmetros:



Classe	Parâmetro
Agent	role
	goal
	backstory
Task	description
	agent
	expected_output

Todos estes são textos que utilizamos para descrever nosso agent e nossa crew, e podemos aproveitar o Chat para nos ajudar a criá-los. Por exemplo, posso usar a seguinte solicitação:

Agora descreva o sistema como agents e tasks, da seguinte forma:

- Cada cargo representa um agente
- Cada agente é descrito por três argumentos:
  - role: Descreve a função do agent dentro do sistema, indicando qual tarefa ele es
  - goal: Define o objetivo principal do agent, ou seja, o que se espera que ele alca
  - backstory: Fornece um contexto que ajuda a moldar a personalidade e a abordagem
- Cada task é descrita por três argumentos:
  - description: Orienta sobre o que a task deve realizar, detalhando as ações especo
  - expected\_output: Descreve o resultado esperado da task, que é utilizado para val
  - agent: Indica qual agent é responsável pela execução da task.

E temos a seguinte resposta:

1. **\*\*Estrategista de Conteúdo\*\***

- **\*\*role:\*\*** Definir a estratégia e elaborar o briefing do artigo.
- **\*\*goal:\*\*** Criar um briefing robusto que inclua objetivos, público-alvo, direcio
- **\*\*backstory:\*\*** Experiente em marketing de conteúdo, analisa temas e direciona e

2. **\*\*Editor de Conteúdo\*\***

- **\*\*role:\*\*** Planejar e estruturar o conteúdo do artigo.
- **\*\*goal:\*\*** Organizar as ideias em um outline claro e lógico, definindo seções, tó
- **\*\*backstory:\*\*** Com vasta experiência em edição, atua como elo entre a estratégia

3. **Redator de Conteúdo**

- **role:** Produzir o rascunho inicial do artigo.
- **goal:** Transformar o briefing e o outline em um texto coeso, engajador e informativo.
- **backstory:** Apaixonado pela escrita, tem habilidade para traduzir ideias complexas em linguagem acessível.

4. **Revisor de Texto**

- **role:** Revisar e aperfeiçoar o artigo.
- **goal:** Garantir que o texto esteja livre de erros gramaticais e ortográficos.
- **backstory:** Detalhista e rigoroso, possui profundo conhecimento em gramática e estilo.

5. **Gestor de Conteúdo**

- **role:** Validar e aprovar o artigo para publicação.
- **goal:** Assegurar que o artigo final esteja alinhado com os objetivos estratégicos.
- **backstory:** Com visão holística do processo de criação, coordena a entrega do conteúdo.

---

**## Tasks**

1. **Task 1 – Elaborar Briefing e Pesquisa**

- **description:** Definir o tema, identificar o público-alvo, estabelecer objetivos claros.
- **expected\_output:** Documento de briefing completo, contendo objetivos claros.
- **agent:** Estrategista de Conteúdo

7. **Task 2 – Estruturar o Conteúdo**

- **description:** Criar um outline que organize os tópicos e seções do artigo, de forma lógica e coerente.
- **expected\_output:** Outline detalhado e organizado do artigo.
- **agent:** Editor de Conteúdo

8. **Task 3 – Redação do Artigo**

- **description:** Produzir o rascunho inicial do artigo com base no briefing e no outline.
- **expected\_output:** Rascunho completo e coerente do artigo, com narrativa fluida.
- **agent:** Redator de Conteúdo

9. **Task 4 – Revisar e Editar o Texto**

- **description:** Revisar o artigo para corrigir erros gramaticais, ortográficos e de estilo.

- **expected\_output:** Versão do artigo revisada e otimizada, sem erros e com fluidez
- **agent:** Revisor de Texto

### 10. **Task 5 – Aprovar e Preparar para Publicação**

- **description:** Validar o artigo final, integrar feedbacks e realizar os ajustes necessários
- **expected\_output:** Artigo final aprovado, formatado e pronto para ser publicado
- **agent:** Gestor de Conteúdo

Perfeito, assim nossa estrutura de CrewAI já está perfeitamente construída! Muito incrível!

Na próxima aula, vamos explorar formas um pouco mais complexas e, consequentemente, mais poderosas de criação de crews: as estruturas YAML!

## 07. Projeto finalizado - Criando uma crew para criar crews

Após alguns ajustes no projeto, chagamos aos seguintes arquivos:

### agents.yaml

```
product_owner:
  role: >
    Definir os requisitos e o escopo do sistema, alinhando as expectativas do projeto com as
    ↪ necessidades do usuário final.
  goal: >
    Garantir que o sistema atenda às necessidades do cliente e que todas as funcionalidades
    ↪ sejam claramente especificadas.
  backstory: >
    Um visionário que entende as demandas do mercado e traduz isso em requisitos práticos. Ele
    ↪ trabalha próximo aos stakeholders para garantir que o produto final gere valor.

desenvolvedor_de_agentes:
  role: >
    Projetar e especificar os agentes necessários no sistema, documentando suas funções e
    ↪ limitações no agents.yaml.
  goal: >
    Definir agentes que consigam executar as tarefas planejadas de maneira eficiente e
    ↪ colaborativa.
  backstory: >
    Um arquiteto detalhista que entende profundamente as capacidades de automação e sabe como
    ↪ distribuir responsabilidades entre os agentes.

desenvolvedor_de_tarefas:
  role: >
    Elaborar as tasks no tasks.yaml, especificando o fluxo de trabalho e as interações entre as
    ↪ tarefas.
  goal: >
    Garantir que todas as tarefas sejam bem definidas, funcionais e alinhadas aos objetivos do
    ↪ sistema.
  backstory: >
    Um planejador estratégico que adora criar fluxos de trabalho otimizados e eficientes,
    ↪ prevendo desafios e criando soluções robustas.

desenvolvedor_de_orquestracao:
  role: >
    Integrar agentes e tasks no crew.py, coordenando a execução das tarefas conforme o fluxo
    ↪ planejado.
  goal: >
    Garantir que o sistema funcione de maneira coesa e harmoniosa, com a execução correta das
    ↪ tasks pelos agentes.
  backstory: >
    Um maestro da automação que sabe exatamente como sincronizar processos complexos,
    ↪ garantindo que tudo funcione em perfeita harmonia.
```

### tasks.yaml

#### definir\_requisitos\_e\_escopo:

```
description: >
  Identificar as funcionalidades necessárias para o sistema e documentar os requisitos de
  ↪ forma clara.
  Sistema solicitado: {especificacoes_sistema}
expected_output: >
  Documento de requisitos detalhado, alinhado com as expectativas dos stakeholders.
agent: product_owner
```

#### especificar\_agentes:

```
description: >
  Definir os agentes no agents.yaml, incluindo seus papéis e permissões dentro do sistema.
  Sistema solicitado: {especificacoes_sistema}
expected_output: >
  Arquivo agents.yaml completo, com agentes especificados e documentados.
  O formato é o seguinte:
  nome_do_agente:
    role: >
      Descreve a função e a especialização do agente dentro da equipe. Aqui você define o
      ↪ papel do agente e sua expertise na aplicação.
    goal: >
      O objetivo individual que guia a tomada de decisões do agente. Este campo define o que o
      ↪ agente busca alcançar com suas ações.
    backstory: >
      Contextualiza o agente, fornecendo informações sobre sua personalidade, experiência e
      ↪ como ele se relaciona com o restante da equipe.
  nome_do_agente_2:
    ...
  Retorne apenas o arquivo solicitado!
agent: desenvolvedor_de_agentes
```

#### definir\_tarefas:

```
description: >
  Criar o tasks.yaml, descrevendo as ações necessárias para cada task, incluindo entradas e
  ↪ saídas.
  Sistema solicitado: {especificacoes_sistema}
expected_output: >
  Arquivo tasks.yaml estruturado com tasks claras e detalhadas.
  O formato é o seguinte:
  nome_da_task:
    description: >
      Uma declaração clara e concisa sobre o que a tarefa envolve. Este campo deve especificar
      ↪ exatamente o que o agente precisa fazer para completar a tarefa. Ele não deve descrever o
      ↪ agente, e sim a task que ele executará.
    expected_output: >
      Uma descrição detalhada do que a conclusão da tarefa deve produzir. O que se espera como
      ↪ resultado final após a tarefa ser executada.
    agent: nome_do_agente (nome conforme arquivo de agentes)
  nome_da_task_2:
    ...
  Retorne apenas o arquivo solicitado!
agent: desenvolvedor_de_tarefas
```

orquestrar\_agentes\_e\_tarefas:

description: >

Implementar o crew.py conectando agentes e tasks, configurando o fluxo de execução.

Sistema solicitado: {especificacoes\_sistema}

expected\_output: >

Arquivo crew.py funcional, com integração completa entre agents.yaml e tasks.yaml.

O código final deve parecer com:

```
```python
from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task
from dotenv import load_dotenv, find_dotenv

_ = load_dotenv(find_dotenv())

@CrewBase
class NomeDaCrew():
    """Teste crew"""

    agents_config = 'config/agents.yaml'
    tasks_config = 'config/tasks.yaml'

    @agent
    def nome_do_primeiro_agente(self) -> Agent:
        return Agent(
            config=self.agents_config['nome_do_primeiro_agente'],
            verbose=True
        )

    @agent
    def nome_do_segundo_agente(self) -> Agent:
        return Agent(
            config=self.agents_config['nome_do_segundo_agente'],
            verbose=True
        )

    @task
    def nome_da_primeira_task(self) -> Task:
        return Task(
            config=self.tasks_config['nome_da_primeira_task'],
        )

    @task
    def nome_da_segunda_task(self) -> Task:
        return Task(
            config=self.tasks_config['nome_da_segunda_task'],
        )

@crew
```

```
def crew(self) -> Crew:
    return Crew(
        agents=self.agents,
        tasks=self.tasks,
        verbose=True,
    )
...
```

Retorne apenas o código python na sua resposta!

**agent:** `desenvolvedor_de_orquestracao`

```
from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task
```

**@CrewBase**

**class** `CrewCreator()`:

*"""CrewCreator crew"""*

`agents_config = 'config/agents.yaml'`

`tasks_config = 'config/tasks.yaml'`

**@agent**

**def** `product_owner(self)` -> `Agent`:

```
    return Agent(
        config=self.agents_config['product_owner'],
        verbose=True
    )
```

**@agent**

**def** `desenvolvedor_de_agentes(self)` -> `Agent`:

```
    return Agent(
        config=self.agents_config['desenvolvedor_de_agentes'],
        verbose=True
    )
```

**@agent**

**def** `desenvolvedor_de_tarefas(self)` -> `Agent`:

```
    return Agent(
        config=self.agents_config['desenvolvedor_de_tarefas'],
        verbose=True
    )
```

**@agent**

**def** `desenvolvedor_de_orquestracao(self)` -> `Agent`:

```
    return Agent(
        config=self.agents_config['desenvolvedor_de_orquestracao'],
        verbose=True
    )
```

**@task**

**def** `definir_requisitos_e_escopo(self)` -> `Task`:

```
    return Task(
        config=self.tasks_config['definir_requisitos_e_escopo'],
    )
```

```
@task
def especificar_agentes(self) -> Task:
    return Task(
        config=self.tasks_config['especificar_agentes'],
        output_file='output/agents.yaml'
    )

@task
def definir_tarefas(self) -> Task:
    return Task(
        config=self.tasks_config['definir_tarefas'],
        output_file='output/tasks.yaml'
    )

@task
def orquestrar_agentes_e_tarefas(self) -> Task:
    return Task(
        config=self.tasks_config['orquestrar_agentes_e_tarefas'],
        output_file='output/crew.py'
    )

@crew
def crew(self) -> Crew:
    """Creates the CrewCreator crew"""
    return Crew(
        agents=self.agents, # Automatically created by the @agent decorator
        tasks=self.tasks, # Automatically created by the @task decorator
        process=Process.sequential,
        verbose=True,
    )
```



## 08. Projeto - Criando uma crew para criar crews

Então vamos colocar a massa em um projeto! A ideia é criamos uma aplicação que já nos gere pronto os arquivos `agents.yaml`, `tasks.yaml` e `crew.py`, para economizar o nosso trabalho ao longo do nosso curso.

Isto já uma mentalidade de quem desenvolve aplicações de IA, se você percebe que haverá uma tarefa que você repetirá diversas vezes, já a automatize agora e, claro, utilizando IA!

Vamos lá, se este projeto for criado vai ser bem legal, pois não mais precisaremos criar nenhum arquivo na mão na hora de desenvolvermos novas crews!

### Desenvolvendo a primeira versão do sistema

Como já vimos, vamos tentar utilizar o chat para criar os nossos `agents` e `tasks`, conforme ensinamos anteriormente. Vamos ver se funcionará!

Primeiro utilizaremos o seguinte prompt no chat:

Dado um processo de criação de uma crew de crewai.

Este consiste em criar arquivos `agents.yaml`, `tasks.yaml` e `crew.py`. Sendo o arquivo `agents.yaml` o seguinte:

Quebre o processo em etapas, com cargos e tarefas executadas por esses cargos, para ser possível.

Ponto inicial: uma sistema de crew que gostamos de desenvolver

Ponto final: os arquivos `agents.yaml`, `tasks.yaml` e `crew.py` criados

Número máximo de etapas: 6

A resposta foi a seguinte:

### \*\*Etapa 1: Definição de Requisitos e Escopo\*\*

- \*\*Cargo Responsável:\*\* \*\*Product Owner\*\*
- \*\*Tarefas:\*\*
  - Definir o objetivo do sistema e as funcionalidades necessárias.
  - Identificar os agentes e tarefas que o sistema precisará.
  - Documentar os requisitos em alto nível para guiar o desenvolvimento.

---

### ### \*\*Etapa 2: Design da Arquitetura e Fluxo de Trabalho\*\*

- **Cargo Responsável:** **Arquiteto de Software**
- **Tarefas:**
  - Estruturar o fluxo de interação entre agentes e tarefas.
  - Definir a arquitetura do sistema para orquestração no `crew.py`.
  - Planejar a organização dos arquivos `agents.yaml`, `tasks.yaml` e `crew.py`.

---

### ### \*\*Etapa 3: Definição dos Agentes\*\*

- **Cargo Responsável:** **Desenvolvedor de Agentes**
- **Tarefas:**
  - Especificar os agentes no arquivo `agents.yaml`, incluindo papéis e permissões.
  - Definir as capacidades e limites de cada agente.
  - Validar se os agentes cobrem todos os requisitos definidos.

---

### ### \*\*Etapa 4: Definição das Tarefas\*\*

- **Cargo Responsável:** **Desenvolvedor de Tarefas**
- **Tarefas:**
  - Descrever as tarefas no arquivo `tasks.yaml`, incluindo entradas, saídas e condições.
  - Garantir que as tarefas estejam alinhadas com os agentes definidos.
  - Testar cada tarefa individualmente para verificar sua funcionalidade.

---

### ### \*\*Etapa 5: Orquestração e Integração\*\*

- **Cargo Responsável:** **Desenvolvedor de Orquestração**
- **Tarefas:**
  - Implementar a lógica de orquestração no `crew.py`, conectando agentes e tarefas.
  - Configurar o fluxo de execução entre as tarefas.

- Realizar testes integrados para garantir a funcionalidade do sistema.

---

### ### \*\*Etapa 6: Testes Finais e Documentação\*\*

- \*\*Cargo Responsável:\*\* \*\*QA e Documentação\*\*
- \*\*Tarefas:\*\*
  - Executar testes de ponta a ponta para validar o sistema.
  - Documentar o funcionamento do `agents.yaml`, `tasks.yaml` e `crew.py`.
  - Reunir feedback e ajustar detalhes conforme necessário.

---

Destes, ficaremos com os agentes 1, 3, 4, e 5.

Vamos continuar com o seguinte prompt:

Mantenha os agentes 1, 3, 4 e 5 para o sistema final e descreva o sistema como agents

- Cada cargo representa um agente
- Cada agente é descrito por três argumentos:
  - role: Descreve a função do agent dentro do sistema, indicando qual tarefa ele es
  - goal: Define o objetivo principal do agent, ou seja, o que se espera que ele alca
  - backstory: Fornece um contexto que ajuda a moldar a personalidade e a abordagem
- Cada task é descrita por três argumentos:
  - description: Orienta sobre o que a task deve realizar, detalhando as ações especo
  - expected\_output: Descreve o resultado esperado da task, que é utilizado para val
  - agent: Indica qual agent é responsável pela execução da task.

E temos a seguinte resposta:

---

## \*\*Agents (Cargos)\*\*

### \*\*Agent 1: Product Owner\*\*

- \*\*role:\*\* Definir os requisitos e o escopo do sistema, alinhando as expectativas do

- **goal:** Garantir que o sistema atenda às necessidades do cliente e que todas as f
- **backstory:** Um visionário que entende as demandas do mercado e traduz isso em re

---

### **Agent 3: Desenvolvedor de Agentes**

- **role:** Projetar e especificar os agentes necessários no sistema, documentando s
- **goal:** Definir agentes que consigam executar as tarefas planejadas de maneira e
- **backstory:** Um arquiteto detalhista que entende profundamente as capacidades de

---

### **Agent 4: Desenvolvedor de Tarefas**

- **role:** Elaborar as tasks no `tasks.yaml`, especificando o fluxo de trabalho e as
- **goal:** Garantir que todas as tarefas sejam bem definidas, funcionais e alinhada
- **backstory:** Um planejador estratégico que adora criar fluxos de trabalho otimiz

---

### **Agent 5: Desenvolvedor de Orquestração**

- **role:** Integrar agentes e tasks no `crew.py`, coordenando a execução das tarefa
- **goal:** Garantir que o sistema funcione de maneira coesa e harmoniosa, com a execo
- **backstory:** Um maestro da automação que sabe exatamente como sincronizar proces

---

## **Tasks**

### **Task 1: Definir Requisitos e Escopo**

- **description:** Identificar as funcionalidades necessárias para o sistema e docun
- **expected\_output:** Documento de requisitos detalhado, alinhado com as expectati
- **agent:** **Product Owner**

---

### ### \*\*Task 2: Especificar Agentes\*\*

- \*\*description:\*\* Definir os agentes no `agents.yaml`, incluindo seus papéis e permissões
- \*\*expected\_output:\*\* Arquivo `agents.yaml` completo, com agentes especificados e suas tarefas
- \*\*agent:\*\* \*\*Desenvolvedor de Agentes\*\*

---

### ### \*\*Task 3: Definir Tarefas\*\*

- \*\*description:\*\* Criar o `tasks.yaml`, descrevendo as ações necessárias para cada tarefa
- \*\*expected\_output:\*\* Arquivo `tasks.yaml` estruturado com tasks claras e detalhadas
- \*\*agent:\*\* \*\*Desenvolvedor de Tarefas\*\*

---

### ### \*\*Task 4: Orquestrar Agentes e Tarefas\*\*

- \*\*description:\*\* Implementar o `crew.py` conectando agentes e tasks, configurando o fluxo de trabalho
- \*\*expected\_output:\*\* Arquivo `crew.py` funcional, com integração completa entre agentes e tarefas
- \*\*agent:\*\* \*\*Desenvolvedor de Orquestração\*\*

---

## Criando o arquivo agents.yaml

Agora vamos criar o arquivo yaml com o seguinte prompt:

Agora crie um arquivo yaml para os agentes, com as seguinte formatação:

```
nome_do_agente:
  role: >
    Role aqui
  goal: >
    Goal aqui
  backstory: >
    Backstory aqui
```

Lembrando que os agentes devem ter um nome em snake case

E temos o arquivo yaml pronto:

```
product_owner:
  role: >
    Definir os requisitos e o escopo do sistema, alinhando as expectativas do projeto com as
    ↳ necessidades do usuário final.
  goal: >
    Garantir que o sistema atenda às necessidades do cliente e que todas as funcionalidades
    ↳ sejam claramente especificadas.
  backstory: >
    Um visionário que entende as demandas do mercado e traduz isso em requisitos práticos. Ele
    ↳ trabalha próximo aos stakeholders para garantir que o produto final gere valor.

desenvolvedor_de_agentes:
  role: >
    Projetar e especificar os agentes necessários no sistema, documentando suas funções e
    ↳ limitações no agents.yaml.
  goal: >
    Definir agentes que consigam executar as tarefas planejadas de maneira eficiente e
    ↳ colaborativa.
  backstory: >
    Um arquiteto detalhista que entende profundamente as capacidades de automação e sabe como
    ↳ distribuir responsabilidades entre os agentes.

desenvolvedor_de_tarefas:
  role: >
    Elaborar as tasks no tasks.yaml, especificando o fluxo de trabalho e as interações entre as
    ↳ tarefas.
  goal: >
    Garantir que todas as tarefas sejam bem definidas, funcionais e alinhadas aos objetivos do
    ↳ sistema.
  backstory: >
    Um planejador estratégico que adora criar fluxos de trabalho otimizados e eficientes,
    ↳ prevendo desafios e criando soluções robustas.

desenvolvedor_de_orquestracao:
  role: >
    Integrar agentes e tasks no crew.py, coordenando a execução das tarefas conforme o fluxo
    ↳ planejado.
  goal: >
    Garantir que o sistema funcione de maneira coesa e harmoniosa, com a execução correta das
    ↳ tasks pelos agentes.
  backstory: >
    Um maestro da automação que sabe exatamente como sincronizar processos complexos,
    ↳ garantindo que tudo funcione em perfeita harmonia.
```

### Criando o arquivo tasks.yaml

Agora vamos para as tasks, com o seguinte prompt:

Agora crie um arquivo yaml para as tasks, com as seguinte formatação:

```
nome_da_task:
  description: >
    Descrição aqui
  expected_output: >
    Saída esperada aqui
  agent: agente_aqui
```

Lembrando que os agentes e tasks devem ter um nome em snake case

E temos a seguinte resposta:

```
definir_requisitos_e_escopo:
  description: >
    Identificar as funcionalidades necessárias para o sistema e documentar os requisitos de
    ↪ forma clara.
  expected_output: >
    Documento de requisitos detalhado, alinhado com as expectativas dos stakeholders.
  agent: product_owner

especificar_agentes:
  description: >
    Definir os agentes no agents.yaml, incluindo seus papéis e permissões dentro do sistema.
  expected_output: >
    Arquivo agents.yaml completo, com agentes especificados e documentados.
  agent: desenvolvedor_de_agentes

definir_tarefas:
  description: >
    Criar o tasks.yaml, descrevendo as ações necessárias para cada task, incluindo entradas e
    ↪ saídas.
  expected_output: >
    Arquivo tasks.yaml estruturado com tasks claras e detalhadas.
  agent: desenvolvedor_de_tarefas

orquestrar_agentes_e_tarefas:
  description: >
    Implementar o crew.py conectando agentes e tasks, configurando o fluxo de execução.
  expected_output: >
    Arquivo crew.py funcional, com integração completa entre agents.yaml e tasks.yaml.
  agent: desenvolvedor_de_orquestracao
```

### Refinando as expected\_output's

Para auxiliar os agentes a produzir saídas válidas, é importante dar exemplos de como queremos estas saídas. Podemos adiciona as formatações esperadas nas tasks especificar\_agentes, definir\_tarefas

e orquestrar\_agentes\_e\_tarefas, da seguinte forma:

definir\_requisitos\_e\_escopo:

description: >

Identificar as funcionalidades necessárias para o sistema e documentar os requisitos.

expected\_output: >

Documento de requisitos detalhado, alinhado com as expectativas dos stakeholders.

agent: product\_owner

especificar\_agentes:

description: >

Definir os agentes no agents.yaml, incluindo seus papéis e permissões dentro do sistema.

expected\_output: >

Arquivo agents.yaml completo, com agentes especificados e documentados.

O formato é o seguinte:

nome\_do\_agente:

role: >

Descreve a função e a especialização do agente dentro da equipe. Aqui você define o papel.

goal: >

O objetivo individual que guia a tomada de decisões do agente. Este campo define o que o agente deve alcançar.

backstory: >

Contextualiza o agente, fornecendo informações sobre sua personalidade, experiências e conhecimentos.

nome\_do\_agente\_2:

...

agent: desenvolvedor\_de\_agentes

definir\_tarefas:

description: >

Criar o tasks.yaml, descrevendo as ações necessárias para cada task, incluindo o contexto e o objetivo.

expected\_output: >

Arquivo tasks.yaml estruturado com tasks claras e detalhadas.

O formato é o seguinte:

nome\_da\_task:

description: >

Uma declaração clara e concisa sobre o que a tarefa envolve. Este campo deve especificar o que precisa ser feito.

expected\_output: >

Uma descrição detalhada do que a conclusão da tarefa deve produzir. O que se espera que seja o resultado final.

agent: nome\_do\_agente (nome conforme arquivo de agentes)



```
nome_da_task_2:  
    ...  
agent: desenvolvedor_de_tarefas
```

orquestrar\_agentes\_e\_tarefas:

description: >

Implementar o crew.py conectando agentes e tasks, configurando o fluxo de execução

expected\_output: >

Arquivo crew.py funcional, com integração completa entre agents.yaml e tasks.yaml

O código final deve parecer com:

```
```python  
from crewai import Agent, Crew, Process, Task  
from crewai.project import CrewBase, agent, crew, task  
from dotenv import load_dotenv, find_dotenv  
  
_ = load_dotenv(find_dotenv())  
  
@CrewBase  
class NomeDaCrew():  
    """Teste crew"""  
  
    agents_config = 'config/agents.yaml'  
    tasks_config = 'config/tasks.yaml'  
  
    @agent  
    def nome_do_primeiro_agente(self) -> Agent:  
        return Agent(  
            config=self.agents_config['nome_do_primeiro_agente'],  
            verbose=True  
        )  
  
    @agent  
    def nome_do_segundo_agente(self) -> Agent:  
        return Agent(  
            config=self.agents_config['nome_do_segundo_agente'],  
            verbose=True  
        )
```

```
@task
def nome_da_primeira_task(self) -> Task:
    return Task(
        config=self.tasks_config['nome_da_primeira_task'],
    )

@task
def nome_da_primeira_task(self) -> Task:
    return Task(
        config=self.tasks_config['revisa_arquivo_yaml_agentes'],
    )

@task
def nome_da_segunda_task(self) -> Task:
    return Task(
        config=self.tasks_config['nome_da_segunda_task'],
    )
@crew
def crew(self) -> Crew:
    return Crew(
        agents=self.agents,
        tasks=self.tasks,
        verbose=True,
    )
...

agent: desenvolvedor_de_orquestracao
```

E já temos a base do nosso projeto. Na próxima aula vamos testá-lo e colocá-lo para rodar!

Até lá!

## 09. Projeto - Refinando a crew para criar crews

Bem-vindos de volta, pessoal! Finalizamos agora os elementos básicos de CrewAI e já vamos começar a pensar na criação de estruturas para produção. Pois nosso objetivo é criar soluções que realmente serão utilizadas; então, quando quisermos oferecê-las ao mundo, elas precisam estar prontas!

Para facilitar este processo, a CrewAI criou recentemente uma nova estrutura de construção de uma Crew, baseada em arquivos YAML. De início, isso pode assustar um pouco, pois há uma complexidade maior na forma como os arquivos são organizados. Porém, eles são mais limpos, modularizados e facilitam a manutenção do projeto.

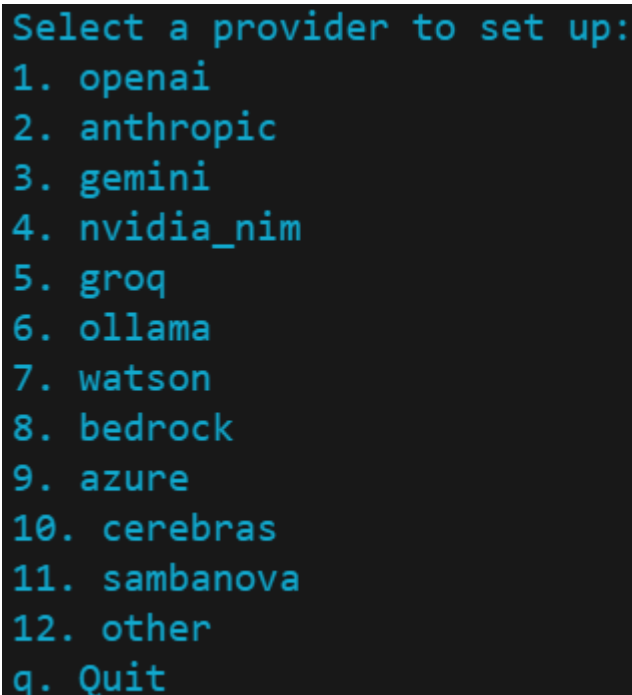
### Iniciando um projeto

Com o CrewAI instalado no seu ambiente, você ganha acesso a alguns comandos criados na biblioteca. Vamos utilizar o comando `create` para iniciar um novo projeto. Para isso, abra um terminal e digite:

```
crewai create crew projeto_teste
```

No exemplo, demos o nome do projeto como `projeto_teste`, mas você pode modificar para um nome que faça sentido para a sua aplicação.

Primeiro, será solicitado que você selecione o provedor de modelos que será utilizado no seu projeto, e você perceberá que temos diversas opções no caso do CrewAI:



```
Select a provider to set up:
1. openai
2. anthropic
3. gemini
4. nvidia_nim
5. groq
6. ollama
7. watson
8. bedrock
9. azure
10. cerebras
11. sambanova
12. other
q. Quit
```

Eu digitei “1” para selecionar a OpenAI, e depois vamos para a seleção do modelo da OpenAI que será utilizado:

```
Select a model to use for Openai:
```

- 1. gpt-4
- 2. gpt-4o
- 3. gpt-4o-mini
- 4. o1-mini
- 5. o1-preview
- q. Quit

```
Enter the number of your choice or 'q' to quit: █
```

Selecionei o gpt-4o-mini para este projeto!

Por último, será solicitado que você forneça uma API KEY do provedor que você selecionou:

```
Enter your OPENAI API key (press Enter to skip):
```

Em geral, eu deixo vazio e forneço a API key manualmente após a estrutura ser criada.

E pronto! A estrutura do projeto estará criada dentro da pasta com o nome do projeto que você forneceu; no nosso caso, projeto\_teste.

## Analizando a estrutura do projeto

A estrutura criada será a seguinte:

```
projeto_teste/
├── .gitignore
├── pyproject.toml
├── README.md
├── .env
├── src/
│   └── projeto_teste/
│       ├── __init__.py
│       ├── main.py
│       ├── crew.py
│       ├── tools/
│       │   └── custom_tool.py
```

```
├── __init__.py
├── config/
│   ├── agents.yaml
│   └── tasks.yaml
```

Temos alguns arquivos principais que precisamos entender para utilizar as Crews nesta estrutura:

Arquivo	Finalidade
agents.yaml	Define seus agentes de IA e seus papéis
tasks.yaml	Configura tarefas e fluxos de trabalho dos agentes
.env	Armazena chaves de API e variáveis de ambiente
main.py	Ponto de entrada do projeto e fluxo de execução
crew.py	Orquestração e coordenação da equipe
tools/	Diretório para ferramentas personalizadas de agentes

### Arquivo .env

Vamos começar analisando o arquivo .env.

```
projeto_teste > ⚙ .env
1  MODEL=gpt-4o-mini
2
```

Você pode perceber que aqui é definido o modelo que o nosso sistema utilizará. Como definimos no início o `gpt-4o-mini`, o comando `create` já preenche o arquivo com o modelo que solicitei.

Neste arquivo, também adicionamos a chave de API do provedor. No final, teremos um arquivo como:

```
MODEL=gpt-4o-mini
OPENAI_API_KEY=MINHA_CHAVE_DE_API
```

No caso da OpenAI, o nome para fornecer a key é `OPENAI_API_KEY`. Este nome varia conforme o provedor!

### Arquivo agents.yaml

Através do arquivo agents.yaml, começamos o desenvolvimento da nossa aplicação. Você lembra que, ao definir um agente, temos três principais parâmetros que precisamos preencher: role, goal e backstory.

Neste arquivo YAML, podemos fazer a definição dos nossos agentes e suas descrições de forma clara e fácil de visualizar, diferente de quando as definições ficavam dentro do código.

As formatações devem seguir o exemplo que vem junto ao criarmos o projeto via linha de comando:

```
researcher:
  role: >
    {topic} Senior Data Researcher
  goal: >
    Uncover cutting-edge developments in {topic}
  backstory: >
    You're a seasoned researcher with a knack for uncovering the latest
    developments in {topic}. Known for your ability to find the most relevant
    information and present it in a clear and concise manner.
```

Você vê que é uma estrutura que utiliza dois pontos e indentação para determinar elementos que pertencem ao elemento anterior. Neste caso, temos um agente researcher sendo definido. Utilizamos os dois pontos e a indentação para evidenciar que os parâmetros role, goal e backstory que serão definidos na sequência fazem parte do researcher declarado anteriormente. O mesmo ocorre com as definições de role, goal e backstory.

Este arquivo, então, eu modificaria para customizar a minha Crew!

### Arquivo tasks.yaml

O mesmo ocorre com o arquivo tasks.yaml. Nele, definimos nossas tasks, também com a formatação devida, com dois pontos e indentação.

Este é o exemplo padrão contido no projeto que a CrewAI cria para nós:

```
research_task:
  description: >
    Conduct a thorough research about {topic}
    Make sure you find any interesting and relevant information given
    the current year is {current_year}.
  expected_output: >
    A list with 10 bullet points of the most relevant information about {topic}
  agent: researcher
```

Podemos ver que adicionamos também o parâmetro agent com o nome do agente que executará esta task, para facilitar o casamento de task e agent.

### Arquivo crew.py

Agora vamos para o arquivo crew.py, onde fazemos a criação da nossa Crew de fato. É ele que importará as definições de agents e tasks, definirá o tipo do processo e combinará as tools aos agents, conforme necessário, o que veremos mais adiante no curso.

Este é o exemplo que vem ao criarmos o projeto inicialmente:

```
from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task

@CrewBase
class ProjetoTeste():
    """ProjetoTeste crew"""

    agents_config = 'config/agents.yaml'
    tasks_config = 'config/tasks.yaml'

    @agent
    def researcher(self) -> Agent:
        return Agent(
            config=self.agents_config['researcher'],
            verbose=True
        )

    @agent
    def reporting_analyst(self) -> Agent:
        return Agent(
            config=self.agents_config['reporting_analyst'],
            verbose=True
        )

    @task
    def research_task(self) -> Task:
        return Task(
            config=self.tasks_config['research_task'],
        )

    @task
    def reporting_task(self) -> Task:
        return Task(
            config=self.tasks_config['reporting_task'],
            output_file='report.md'
        )

    @crew
    def crew(self) -> Crew:
        """Creates the ProjetoTeste crew"""

        return Crew(
            agents=self.agents,
            tasks=self.tasks,
            process=Process.sequential,
```

```
        verbose=True,  
    )
```

Você pode nunca ter se deparado com uma estrutura como a classe `ProjetoTeste` em Python, mas é bem simples! Essa classe foi criada para unificar a configuração da sua crew, e você verá que não precisará modificá-la drasticamente.

Além das classes, temos decoradores, aquelas palavras com um `@` na frente. Estes decoradores também são padrão da bibliotecas mas de super fácil utilização!

---

E para tudo ficar bem claro, vou deixar uma breve explicação de o que é classe e decoradores para quem não tem familiaridade com esses termos.

### O que é uma classe?

Em Python, uma classe é como um molde que define um tipo de objeto. Dentro de uma classe, podemos definir propriedades (dados) e métodos (funções) que descrevem o comportamento desse objeto. No caso da classe `ProjetoTeste`, estamos definindo como criar e gerenciar uma crew de agentes e tarefas.

### O que são decoradores?

Em termos simples, decoradores são uma forma de modificar o comportamento de uma função, método ou classe sem alterar seu código interno. No contexto da nossa classe `ProjetoTeste`, usamos decoradores para marcar quais métodos se tornarão agents, tasks e o método que define a crew.

---

Por exemplo, ao criar o agente `researcher`, usamos o decorador `@agent` antes da definição do método. Isso indica que o método subsequente será um agent. Dentro do método, simplesmente referenciamos o nome do agente, conforme definido no arquivo `agents.yaml`, utilizando `self.agents_config['researcher']`. Isso garante que todas as configurações do agente sejam carregadas automaticamente a partir do arquivo YAML.

O processo para definir as tasks é bastante semelhante. Neste caso, usamos o decorador `@task`, e assim como fizemos com os agents, configuramos o método para apontar para a variável `tasks_config`, onde estão definidas as tasks no arquivo `tasks.yaml`.

Agora, é importante notar que você precisará criar um número de métodos equivalente à quantidade de agents e tasks que a crew terá. Não se preocupe! Você pode facilmente fazer cópias dos métodos existentes, alterar o nome (por exemplo, de `researcher` para `reporting_analyst`) e



ajustar a configuração que corresponde ao novo agente ou task. Se você definiu um agente chamado `reporting_analyst` no arquivo `agents.yaml`, basta copiar o método `researcher`, renomeá-lo e garantir que a configuração referencie `self.agents_config['reporting_analyst']`. O mesmo se aplica às `tasks`.

E pronto! Assim, você pode configurar sua `crew` de forma simples e eficiente.

### Arquivo `main.py`

Antes de começarmos, vamos ignorar as funções `train`, `replay` e `test`, pois não será conteúdo deste curso. O foco será na função `run()`, que é crucial para a execução da sua `crew`.

Na função `run()`, o mais importante a perceber é que é ali que passamos os parâmetros necessários para a nossa `crew`. Esses parâmetros são representados por um dicionário chamado `inputs`. No exemplo, estamos passando um tópico ('AI LLMs') e o ano atual, que é obtido dinamicamente através de `datetime.now().year`.

Após definir os `inputs`, a `crew` é executada com o método `kickoff(inputs=inputs)`. Esse método inicia o processo e utiliza os dados fornecidos para orquestrar as interações entre os agentes e `tasks`.

Portanto, o ponto central da função `run()` é a passagem dos parâmetros e a execução da `crew`, que são essenciais para o funcionamento da sua aplicação.

### Conclusão

Neste módulo, tivemos uma visão geral da estrutura baseada em arquivos YAML que configura a nossa aplicação com o CrewAI. Ao modificar essa estrutura para criar seu próprio projeto, o processo deve começar pelo arquivo `agents.yaml`, onde definimos os agentes e suas funções. Em seguida, passamos para o arquivo `tasks.yaml`, onde configuramos as tarefas que cada agente executará.

Depois disso, ajustamos o arquivo `crew.py` para orquestrar a interação entre os agentes e tarefas, utilizando decoradores para estabelecer a configuração da `crew`. Por fim, finalizamos com o arquivo `main.py`, que será responsável pela execução do nosso projeto, permitindo a passagem de parâmetros e o controle do fluxo da aplicação.

Nas próximas aulas, estaremos mais envolvidos em como modificar essa estrutura, permitindo que você crie um projeto personalizado a partir dela. Aproveite este conhecimento, pois ele será fundamental para o desenvolvimento de suas aplicações de inteligência artificial com eficiência e clareza!

## 10. Observabilidade com AgentOps

Quando criamos aplicações de IA, trabalhamos com muito texto, tanto na criação de prompts dos nossos agentes quanto nas respostas que eles geram. Estes textos representam a inteligência e os resultados da nossa aplicação, portanto, são fundamentais. A nossa capacidade de entendê-los e de analisá-los vai determinar o quão rápido conseguiremos criar e depurar nossas aplicações. Mas como podemos estabelecer uma forma fácil e prática para analisar todos os prompts utilizados no nosso sistema, assim como as respostas dos nossos agentes e a sequência de interações entre eles, especialmente em sistemas multiagentes? Para isso, podemos utilizar a ferramenta AgentOps, que nos proporciona observabilidade.

### O que é Observabilidade?

Observabilidade é uma abordagem que nos permite entender o comportamento de sistemas complexos, como agentes conversacionais, ao registrar e analisar dados sobre sua performance. Ela nos permite responder questões como: Como meus agentes estão interagindo com os usuários? Como estão utilizando ferramentas externas e APIs? Quais métricas são relevantes para avaliar a eficácia e eficiência dos agentes?

### Introduzindo o AgentOps

AgentOps é um produto independente do CrewAI que fornece uma solução abrangente de observabilidade para agentes. Com o AgentOps, você tem acesso a um painel que permite monitorar o custo, uso de tokens, latência, falhas de agentes e muito mais.

### Principais Funcionalidades do AgentOps

1. **Gerenciamento e Rastreamento de Custos:** Monitore os gastos com provedores de modelos fundacionais, garantindo que você esteja ciente dos custos associados ao uso de suas APIs.
2. **Análises de Replay:** Assista a gráficos de execução passo a passo dos agentes, permitindo que você compreenda o fluxo de operações em detalhes.
3. **Deteção de Pensamento Recursivo:** Identifique quando os agentes caem em loops infinitos, possibilitando intervenções imediatas.
4. **Relatórios Personalizados:** Crie análises personalizadas sobre o desempenho do agente, ajustando as métricas que são mais relevantes para o seu caso de uso.

5. **Interface do Pannel:** O painel do AgentOps proporciona uma visão geral de estatísticas altas sobre agentes em desenvolvimento e produção, além de permitir um drilldown para ver interações em tempo real.

## Como Utilizar o AgentOps

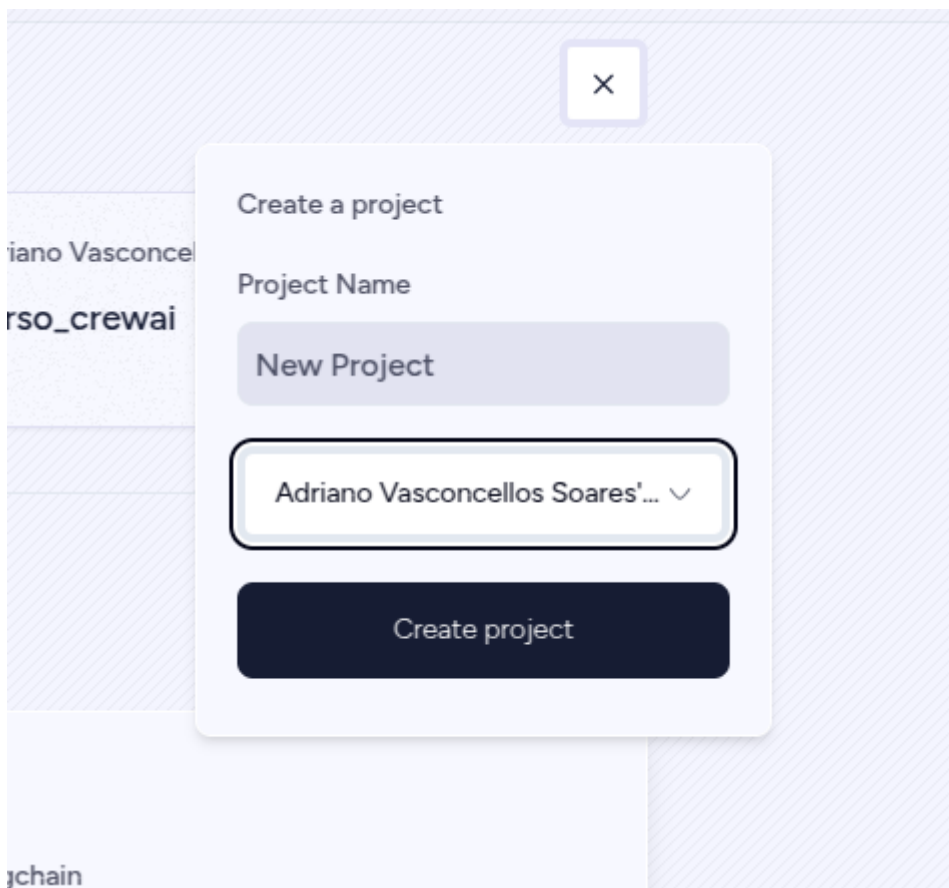
Para começar a usar o AgentOps, você deve realizar algumas etapas simples:

### Crie uma conta na AgentOps

Acesse o seguinte link para fazer o cadastro: <https://app.agentops.ai/signin>

### Crie um novo projeto

Entre na [página de projetos](#) e crie um novo projeto.



### Salve sua chave de API

Salve a chave de API apresentada durante a criação do projeto.

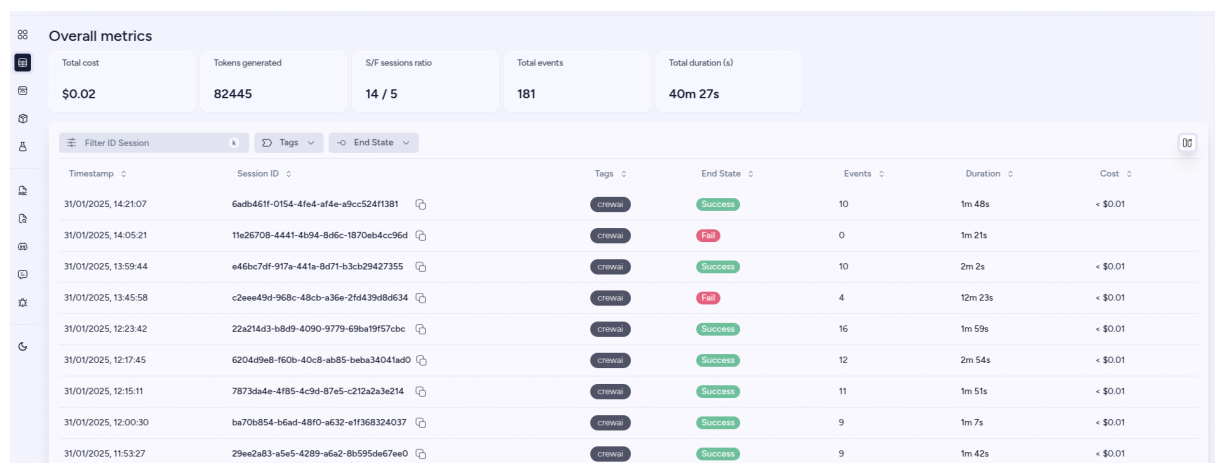


Para isso, basta adicionar o seguinte código antes de rodar a sua Crew:

```
import agentops
agentops.init()
```

### O painel do AgentOps

O painel do AgentOps é uma ferramenta poderosa que oferece uma visualização abrangente do comportamento dos agentes. Após configurar o AgentOps, cada execução do seu programa é registrada como uma sessão, e todos os dados relevantes (como prompts, execuções e eventos) são gravados automaticamente. Você pode acessar detalhes sobre cada sessão, como tempo de execução total, redesenhos de chamadas LLM e muito mais.

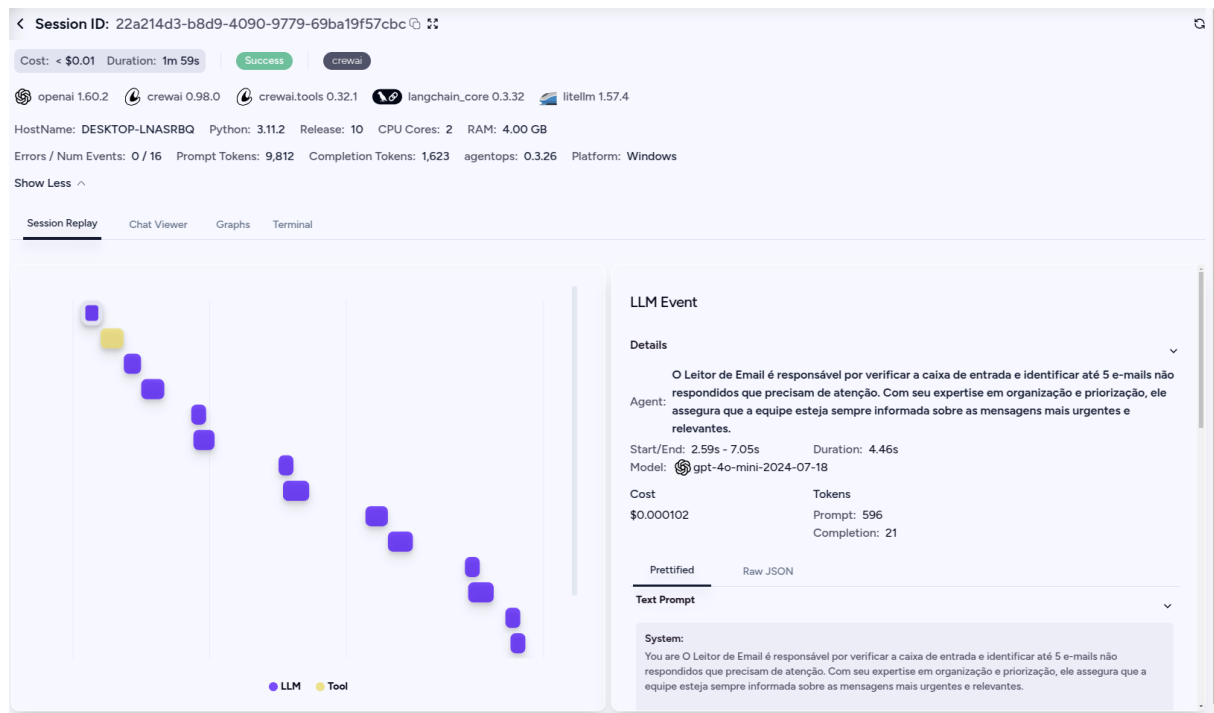


The screenshot displays the AgentOps dashboard. At the top, there are five summary cards for overall metrics: Total cost (\$0.02), Tokens generated (82445), S/F sessions ratio (14 / 5), Total events (181), and Total duration (40m 27s). Below these is a table of sessions with columns for Timestamp, Session ID, Tags, End State, Events, Duration, and Cost. The sessions listed are from 31/01/2025, with various IDs and states (Success or Fail).

Timestamp	Session ID	Tags	End State	Events	Duration	Cost
31/01/2025, 14:21:07	6ad1b461f-0154-4fe4-af4e-a9cc524f1381	crewai	Success	10	1m 48s	< \$0.01
31/01/2025, 14:05:21	11e26708-4441-4b94-8d6c-1870eb4cc96d	crewai	Fail	0	1m 21s	
31/01/2025, 13:59:44	e46bc7df-917a-441a-8d71-b3cb29427355	crewai	Success	10	2m 2s	< \$0.01
31/01/2025, 13:45:58	c2eee49d-968c-48cb-a36e-2fd439d8d634	crewai	Fail	4	12m 23s	< \$0.01
31/01/2025, 12:23:42	22a214d3-b8d9-4090-9779-69ba19f57cbc	crewai	Success	16	1m 59s	< \$0.01
31/01/2025, 12:17:45	6204d9e8-f60b-40c8-ab85-beba34041ad0	crewai	Success	12	2m 54s	< \$0.01
31/01/2025, 12:15:11	7873da4e-4f85-4c9d-87e5-c212a2a3e214	crewai	Success	11	1m 51s	< \$0.01
31/01/2025, 12:00:30	ba70b854-b6ad-48f0-a632-e1f368324037	crewai	Success	9	1m 7s	< \$0.01
31/01/2025, 11:53:27	29ee2a83-a5e5-4289-a6a2-8b595de67ee0	crewai	Success	9	1m 42s	< \$0.01

### Análise de Sessões

O painel oferece um “Session Drawer” que lista todas as sessões anteriormente registradas e dados úteis sobre cada uma. O “Session Waterfall” apresenta uma visualização em tempo real das chamadas de LLM, eventos de ação, chamadas de ferramentas e erros, permitindo que você obtenha detalhes específicos sobre eventos selecionados.



## Conclusão

Quando falamos de observabilidade, estamos tocando em algo super importante para quem quer criar aplicações inteligentes com o CrewAI. Ter essa capacidade de observar e entender o que está acontecendo com seus agentes é fundamental para garantir que tudo esteja rodando direitinho. O AgentOps é uma ótima opção com suas ferramentas de rastreamento, replay e análise, ele se torna um aliado indispensável para quem deseja desenvolver sistemas que realmente funcionem bem.

Nas próximas aulas, vamos entender juntos como você pode usar essas habilidades de observabilidade em projetos práticos. Vamos aprender a melhorar a performance dos seus agentes e, consequentemente, a experiência de quem interage com eles.

## 11. Tools de CrewAI

Vocês devem se lembrar das duas principais características de um agente: raciocinar e agir através de ferramentas.

Até agora, mostramos como criar um sistema multiagentes, adicionando passos e momentos para que ele reflita, o que por si só já aumenta muito a capacidade de uma aplicação. Mas ainda não falamos nada do segundo elemento fundamental: as ações!

### Como um modelo age?

Os modelos necessitam de um meio para conseguir agir. Lembrando que eles são modelos de linguagem; portanto, recebem texto como input e devolvem texto como output. Os modelos, por si só, não têm a habilidade de executar uma ação que não seja gerar texto. Mas como então fornecemos a eles uma maior capacidade de ação, como acessar um banco de dados ou enviar um e-mail ou mensagem no celular? É aí que entra o conceito de desenvolvimento de ferramentas, mais conhecidas pelo termo em inglês: tools!

### O que são as tools?

As tools são funções construídas através da programação que são descritas e fornecidas aos modelos de linguagem. Como falamos anteriormente, o modelo por si só não consegue executar uma tool, mas ele é capaz de entender que existe uma tool dentro do sistema que ele pode solicitar que seja acionada quando necessário. A tool roda dentro do sistema que está chamando o modelo de linguagem. No nosso caso, seria o nosso próprio computador.

O modelo entende que existe uma tool através da descrição que criamos dela. Ele percebe sua utilidade através da descrição que é enviada junto ao prompt, e, ao ver a necessidade de buscar uma informação externa ou atuar de alguma forma, ele se utiliza das tools previamente criadas para isso.

### Os dois principais tipos de tools

Eu gosto de separar as tools em duas categorias: tools de busca de informação e tools de atuação.

#### Tools de captura de informação

Uma das grandes limitações dos modelos de linguagem é que sua informação é limitada aos dados de treinamento. Toda nova informação que surja após o treinamento do modelo de linguagem não

está disponível dentro de sua inteligência. É como se os modelos vivessem sempre há um ano atrás do dia de hoje, ignorando completamente a informação nova que ocorreu neste último ano. Por isso, quando tentamos gerar código para bibliotecas novas, como a própria CrewAI, os códigos estão sempre desatualizados, pois o modelo ignora toda evolução da biblioteca no último ano.

Para contornar essa limitação, podemos criar ferramentas de busca de informações. Alguns exemplos incluem:

- Ferramentas de web crawl para buscar na internet informações recentes.
- Ferramentas de acesso a bancos de dados, oferecendo ao modelo acesso a informações em diversos tipos de banco de dados, como SQL, Postgres, MongoDB, etc.
- Ferramentas de leitura de arquivos de texto, arquivos PDF, CSV, etc.
- Ferramentas de acesso a informações de APIs diversas, como APIs de clima, trânsito, tráfego aéreo, etc.

### Tools de atuação

Agora estamos falando de dar mais autonomia aos modelos. Queremos que nossos modelos possam gerar soluções além da geração de texto — queremos que eles interajam, atuem e modifiquem seu entorno. É aqui que entram as tools de atuação, capazes de mudar o estado de aplicações e sistemas fora do escopo da aplicação. São ferramentas que atuam no mundo real.

Alguns exemplos incluem:

- Ferramentas para enviar um e-mail, uma mensagem SMS, ou uma mensagem no WhatsApp ou Telegram.
- Ferramentas para adicionar ou deletar dados de diversos bancos de dados, como SQL, Postgres, MongoDB, etc.
- Ferramentas para criar e deletar arquivos em um computador.
- Ferramentas para criação de imagens, áudio, etc.

### Como usar tools?

Para usar as Tools no CrewAI, você precisa começar instalando o pacote de ferramentas adicionais. Basta rodar o seguinte comando no seu terminal:

```
pip install 'crewai[tools]'
```

Aqui está um exemplo de como você pode usar algumas Tools no seu código:

```
import os
from crewai import Agent, Task, Crew
```



```
from crewai_tools import (
    DirectoryReadTool,
    FileReadTool,
    SerperDevTool,
    WebsiteSearchTool
)

# Configurando as chaves de API
os.environ["SERPER_API_KEY"] = "Sua Chave" # Chave da API serper.dev
os.environ["OPENAI_API_KEY"] = "Sua Chave"

# Instanciando as ferramentas
docs_tool = DirectoryReadTool(directory='./blog-posts')
file_tool = FileReadTool()
search_tool = SerperDevTool()
web_rag_tool = WebsiteSearchTool()

# Criando agentes
researcher = Agent(
    role='Market Research Analyst',
    goal='Provide up-to-date market analysis of the AI industry',
    backstory='An expert analyst with a keen eye for market trends.',
    tools=[search_tool, web_rag_tool],
    verbose=True
)

writer = Agent(
    role='Content Writer',
    goal='Craft engaging blog posts about the AI industry',
    backstory='A skilled writer with a passion for technology.',
    tools=[docs_tool, file_tool],
    verbose=True
)

# Definindo tarefas
research = Task(
    description='Research the latest trends in the AI industry and provide a summary.',
    expected_output='A summary of the top 3 trending developments in the AI industry with a
    ↪ unique perspective on their significance.',
    agent=researcher
)

write = Task(
    description='Write an engaging blog post about the AI industry, based on the research
    ↪ analyst's summary. Draw inspiration from the latest blog posts in the directory.',
    expected_output='A 4-paragraph blog post formatted in markdown with engaging, informative,
    ↪ and accessible content, avoiding complex jargon.',
    agent=writer,
    output_file='blog-posts/new_post.md' # O post final será salvo aqui
)

# Montando uma crew
crew = Crew(
    agents=[researcher, writer],
```

```
tasks=[research, write],
verbose=True,
planning=True, # Habilitar o recurso de planejamento
)

# Executando as tarefas
crew.kickoff()
```

### ### Analisando o código

Vamos analisar passo-a-passo, compreendendo os elementos que se referem a adição das

#### #### Importação das Tools

```
```python
from crewai_tools import (
    DirectoryReadTool,
    FileReadTool,
    SerperDevTool,
    WebsiteSearchTool
)
```

- Aqui estamos importando várias classes de ferramentas disponíveis no módulo `crewai_tools`. Cada uma dessas classes representa uma Tool específica que pode ser utilizada pelos agentes. Por exemplo:
  - **DirectoryReadTool**: Essa ferramenta é utilizada para ler arquivos de um diretório específico. É útil quando você precisa acessar e processar vários arquivos que estão organizados em uma pasta.
  - **FileReadTool**: Esta Tool permite que os agentes leiam o conteúdo de arquivos, sejam eles TXT, PDF, CSV, entre outros. É ideal para operações que requerem a extração de informações de documentos.
  - **SerperDevTool**: Esta ferramenta interage com a API Serper.dev, possibilitando que os agentes busquem informações na web. É extremamente útil para obter dados atualizados e relevantes que não estão incorporados na base de conhecimento do modelo.
  - **WebsiteSearchTool**: Similar à SerperDevTool, esta ferramentas permite a busca de informações em páginas da web, ampliando ainda mais a capacidade dos agentes de encontrar dados externos.

A biblioteca CrewAI possui diversas Tools diferentes já criadas e prontas para uso, cada uma com funcionalidades específicas que ajudam a melhorar a interação dos agentes com o mundo externo. Isso

permite que você escolha e utilize as ferramentas mais adequadas para as necessidades do seu projeto, facilitando o desenvolvimento e a implementação de soluções mais robustas.

### Instanciação das Tools

```
docs_tool = DirectoryReadTool(directory='./blog-posts')
file_tool = FileReadTool()
search_tool = SerperDevTool()
web_rag_tool = WebsiteSearchTool()
```

- Aqui, criamos instâncias das Tools.
  - `DirectoryReadTool` recebe um parâmetro (`directory`) que especifica qual pasta deve ser lida para buscar arquivos.
  - `FileReadTool`, `SerperDevTool` e `WebsiteSearchTool` são instanciados sem parâmetros necessários. Isso significa que eles têm configurações padrão definidas na própria classe.

### Adicionando Tools aos Agentes

```
researcher = Agent(
    role='Market Research Analyst',
    goal='Provide up-to-date market analysis of the AI industry',
    backstory='An expert analyst with a keen eye for market trends.',
    tools=[search_tool, web_rag_tool],
    verbose=True
)
```

- Ao criar o agente `researcher`, estamos associando as Tools `search_tool` e `web_rag_tool` a ele. Isso significa que, durante sua execução, esse agente poderá utilizar essas ferramentas para realizar tarefas relacionadas a busca de informações e interação com a web.
- O mesmo se aplica ao agente `writer`, que tem `docs_tool` e `file_tool` como suas ferramentas.

### Tools disponibilizadas pela CrewAI

Uma das grandes vantagens da biblioteca CrewAI é a variedade de Tools já construídas pelos desenvolvedores, que facilitam o trabalho com os agentes e expandem suas capacidades. Essas ferramentas são projetadas para lidar com diferentes tipos de tarefas e interações, tornando sua aplicação mais robusta e eficiente. Vamos dar uma olhada nas principais Tools disponíveis:

Tool	Descrição
<b>BrowserbaseLoadTool</b>	Uma ferramenta para interagir e extrair dados de navegadores da web.
<b>CodeDocsSearchTool</b>	Uma Tool otimizada para busca através da documentação de código e documentos técnicos relacionados.
<b>CodeInterpreterTool</b>	Uma ferramenta para interpretar código Python.
<b>ComposioTool</b>	Permite o uso de ferramentas Composio.
<b>CSVSearchTool</b>	Uma Tool projetada para buscar dentro de arquivos CSV, feita especialmente para lidar com dados estruturados.
<b>DALL-E Tool</b>	Uma ferramenta para gerar imagens usando a API DALL-E.
<b>DirectorySearchTool</b>	Uma Tool para busca dentro de diretórios, útil para navegar por sistemas de arquivos.
<b>DOCXSearchTool</b>	Uma Tool voltada para buscar dentro de documentos DOCX, ideal para processamento de arquivos do Word.
<b>DirectoryReadTool</b>	Facilita a leitura e o processamento das estruturas de diretórios e seu conteúdo.
<b>EXASearchTool</b>	Uma ferramenta projetada para realizar buscas exaustivas em várias fontes de dados.
<b>FileReadTool</b>	Permite a leitura e extração de dados de arquivos, suportando diversos formatos de arquivo.
<b>FirecrawlSearchTool</b>	Uma ferramenta para buscar páginas da web usando Firecrawl e retornar os resultados.
<b>FirecrawlCrawlWebsiteTool</b>	Uma ferramenta para rastrear páginas da web usando Firecrawl.
<b>FirecrawlScrapeWebsiteTool</b>	Uma ferramenta para raspar conteúdos de páginas da web usando Firecrawl e retornar seu conteúdo.
<b>GithubSearchTool</b>	Uma Tool para buscar dentro de repositórios do GitHub, útil para pesquisa de código e documentação.
<b>SerperDevTool</b>	Uma ferramenta especializada para fins de desenvolvimento, com funcionalidades específicas em desenvolvimento.
<b>TXTSearchTool</b>	Uma Tool focada na busca dentro de arquivos de texto (.txt), adequada para dados não estruturados.
<b>JSONSearchTool</b>	Uma ferramenta projetada para busca dentro de arquivos JSON, atendendo a manipulação de dados estruturados.

Tool	Descrição
<b>LlamaIndexTool</b>	Permite o uso das ferramentas LlamaIndex.
<b>MDXSearchTool</b>	Uma Tool projetada para busca dentro de arquivos Markdown (MDX), útil para documentação.
<b>PDFSearchTool</b>	Uma ferramenta para busca dentro de documentos PDF, ideal para processamento de documentos digitalizados.
<b>PGSearchTool</b>	Uma Tool otimizada para busca dentro de bancos de dados PostgreSQL, adequada para consultas em bancos de dados.
<b>Vision Tool</b>	Uma ferramenta para gerar imagens usando a API DALL-E.
<b>RagTool</b>	Uma Tool geral de RAG capaz de lidar com várias fontes e tipos de dados.
<b>ScrapeElementFromWebsiteTool</b>	Extrai elementos específicos de sites, útil para extração de dados direcionada.
<b>ScrapeWebsiteTool</b>	Facilita a raspagem de sites inteiros, ideal para coleta abrangente de dados.
<b>WebsiteSearchTool</b>	Uma Tool para buscar conteúdo de websites, otimizada para extração de dados da web.
<b>XMLSearchTool</b>	Uma ferramenta projetada para busca dentro de arquivos XML, adequada para formatos de dados estruturados.
<b>YoutubeChannelSearchTool</b>	Tool para buscar dentro de canais do YouTube, útil para análise de conteúdo em vídeo.
<b>YoutubeVideoSearchTool</b>	Uma ferramenta que visa buscar dentro de vídeos do YouTube, ideal para extração de dados em vídeo.

As Tools de CrewAI desempenham um papel essencial em como seus agentes operam, permitindo a execução de ações que vão muito além da simples geração de texto. Com o conhecimento sobre as duas principais categorias de ferramentas — aquelas que buscam informações e aquelas que atuam — você agora tem as bases para integrar essas funcionalidades em suas aplicações.

Além disso, a biblioteca `crewai_tools` já vem com uma variedade de Tools prontas para uso, cada uma projetada para ajudar em tarefas específicas, como leitura de arquivos, busca na web e interação com APIs. Essa diversidade permite que você escolha as ferramentas mais adequadas para as necessidades do seu projeto, facilitando o desenvolvimento e aumentando a eficiência dos seus agentes.

Na próxima aula, teremos a oportunidade de explorar algumas dessas Tools em detalhes e ver como integrá-las nas nossas aplicações para obter resultados ainda melhores. Fique atento e prepare-se

para expandir suas habilidades com o CrewAI!

## 12. Adicionando tools ao meu projeto

E vocês lembram do criador de blog posts? Aquela primeira crew que criamos no início do nosso curso!

Agora que sabemos utilizar tools, podemos aumentar as capacidades da nossa crew inicial, adicionando algumas ferramentas para potencializar o seu trabalho.

Fiz este código atualizado para vocês testarem! ## agents.yaml

pesquisador\_de\_conteudo:

role: >

O Pesquisador de Conteúdo é especialista em extração de informações relevantes através de ferramentas de busca. Utiliza a SerperDevTool para investigar profundamente os tópicos, identificando tendências, questões relevantes e materiais de referência que servirão como base sólida para o fluxo criativo da equipe.

goal: >

O objetivo do Pesquisador de Conteúdo é garantir que a equipe tenha acesso a informações necessárias e atualizadas sobre um tópico específico, permitindo que o planejamento e a criação do conteúdo sejam fundamentados em dados concretos e relevantes.

backstory: >

Sempre foi apaixonado pela busca do conhecimento e pela arte de contar histórias. Formado em Jornalismo, trabalhou anteriormente em redações de revistas e é conhecido por sua curiosidade insaciável. Seu foco é em sempre trazer à tona novos ângulos e aprofundar-se nos assuntos, colaborando com a equipe para criar conteúdos únicos e informativos.

planejador\_de\_conteudo:

role: >

O Planejador de Conteúdo é responsável por organizar as informações coletadas pelo pesquisador e transformá-las em um guia estratégico. Ele utiliza a ferramenta ScrapeWebsiteTool para fazer scraping de sites, elabora um briefing detalhado e define o formato e a estrutura do conteúdo a ser desenvolvido.

goal: >

O objetivo do Planejador de Conteúdo é criar um esboço que não apenas represente as melhores práticas da criação de conteúdo, mas que também esteja alinhado aos interesses do público-alvo, garantindo que o texto final seja atraente e informativo.

backstory: >

Com um mestrado em Comunicação Digital, o Planejador de Conteúdo desenvolveu uma habilidade aguçada para estruturar informações complexas de maneira que elas sejam acessíveis e envolventes. Seu papel na equipe é o de um maestro, onde combina dados de pesquisa com insights criativos, tudo isso enquanto colabora intimamente com o escritor e revisor.

escritor\_do\_blog:

role: >

O Escritor do Blog é o criador do conteúdo baseado nas diretrizes e briefings fornecidos pelo Planejador de Conteúdo. Ele transforma informações e ideias em textos coesos, claros e envolventes, que se conectam com os leitores e comunicam a mensagem desejada de forma eficaz.

goal: >

O objetivo do Escritor do Blog é desenvolver artigos que atraiam, informem e mantenham a atenção do público, utilizando uma linguagem acessível que ressoe com a audiência, contribuindo para o sucesso geral da equipe no ambiente digital.

backstory: >

Apaixonado pela escrita desde criança, o Escritor do Blog decidiu fazer da sua paixão sua carreira. Com experiência em diversos estilos de redação, ele desenvolveu uma voz única que combina criatividade e técnica. Ele acredita no poder das palavras e trabalha incansavelmente para assegurar que o conteúdo final não seja apenas informativo, mas também inspirador.

**revisor\_do\_conteudo:**

**role:** >

O Revisor do Conteúdo é encarregado de assegurar a qualidade do texto final, revisando para consistência, clareza e precisão. Ele também agrega valor ao conteúdo, adicionando informações extras e refinando a mensagem antes de ser publicada, usando a ferramenta FileWriterTool para salvar o arquivo.

**goal:** >

O objetivo do Revisor do Conteúdo é entregar um produto final que seja não apenas livre de erros, mas que também tenha um nível elevado de qualidade e mérito, garantindo que todos os artigos estejam prontos para impressionar o público e reforçar a credibilidade do blog.

**backstory:** >

Com vasta experiência em edição e revisão em editoras renomadas, o Revisor do Conteúdo é alguém que valoriza a excelência. Sua atenção aos detalhes é implacável, e ele se considera o guardião da qualidade da equipe, sempre buscando maneiras de enriquecer o conteúdo. O Revisor acredita firmemente que todo texto pode sempre ser melhorado e se dedica a cada palavra para alcançar esse objetivo.

## tasks.yaml

**pesquisa\_conteudo:**

**description:** >

O pesquisador de conteúdo utiliza a ferramenta SerperDevTool para buscar informações relevantes sobre o tópico {topico}. Ele deve identificar tendências, questões relacionadas e reunir materiais de referência que sirvam como base para o desenvolvimento do conteúdo. Lembrando que estamos no ano de 2025.

**expected\_output:** >

Um relatório de pesquisa que contenha um resumo das informações coletadas, incluindo links para fontes relevantes, tópicos de interesse e dados importantes que apoiarão as próximas etapas de planejamento e criação de conteúdo.

**agent:** pesquisador\_de\_conteudo

**planejamento\_conteudo:**

**description:** >

O planejador de conteúdo organiza as informações do relatório do pesquisador e utiliza a ferramenta ScrapeWebsiteTool para fazer scraping dos sites, criando um briefing detalhado que define o conteúdo que será desenvolvido, incluindo formato e estrutura, gerando bons briefings para o tópico: {topico}.

**expected\_output:** >

Um briefing completo que delineia claramente a estrutura do conteúdo, os pontos principais a serem abordados, e uma lista de fontes ou materiais que serão utilizados no desenvolvimento do artigo.

**agent:** planejador\_de\_conteudo

**escrita\_blog\_post:**

**description:** >



```
O escritor do blog utiliza o briefing desenvolvido pelo planejador de conteúdo para criar
↪ um artigo completo do topico de {topico}, garantindo que todas as informações sejam
↪ apresentadas de maneira coesa e que o texto atenda aos objetivos da equipe.
expected_output: >
    O artigo de blog escrito, que deve ser envolvente, claro e informativo, abordando todos os
↪ pontos discutidos no briefing e pronto para a revisão final.
agent: escritor_do_blog

revisao_conteudo:
    description: >
        O revisor do conteúdo revisa o artigo escrito para assegurar que não haja erros de
↪ gramática, ortografia ou clareza. Ele também adiciona informações relevantes que possam
↪ enriquecer o texto.
    expected_output: >
        Um documento revisado e polido, livre de erros, que inclui todas as melhorias feitas e que
↪ está formatado corretamente para publicação, pronto para ser disponibilizado ao público.
    agent: revisor_do_conteudo

salva_conteudo:
    description: >
        Salva o conteúdo utilizando a tool FileWriterTool, criando um nome de arquivo que faça
↪ sentido com relação ao conteúdo do texto.
    expected_output: >
        Arquivo salvo.
    agent: revisor_do_conteudo
```

### crew.py

```
from crewai import Agent, Crew, Process, Task
from crewai_tools import SerperDevTool, ScrapeWebsiteTool, FileWriterTool
from crewai.project import CrewBase, agent, crew, task
from dotenv import load_dotenv, find_dotenv

_ = load_dotenv(find_dotenv())

serper = SerperDevTool()
scraper = ScrapeWebsiteTool()
file_write = FileWriterTool()

@CrewBase
class ContentCreationCrew():
    """Crew for content creation process"""

    agents_config = 'config/agents.yaml'
    tasks_config = 'config/tasks.yaml'

    @agent
    def pesquisador_de_conteudo(self) -> Agent:
        return Agent(
            config=self.agents_config['pesquisador_de_conteudo'],
            verbose=True,
            tools=[serper]
```

```
)

@agent
def planejador_de_conteudo(self) -> Agent:
    return Agent(
        config=self.agents_config['planejador_de_conteudo'],
        verbose=True,
        tools=[scraper]
    )

@agent
def escritor_do_blog(self) -> Agent:
    return Agent(
        config=self.agents_config['escritor_do_blog'],
        verbose=True
    )

@agent
def revisor_do_conteudo(self) -> Agent:
    return Agent(
        config=self.agents_config['revisor_do_conteudo'],
        verbose=True,
    )

@task
def pesquisa_conteudo(self) -> Task:
    return Task(
        config=self.tasks_config['pesquisa_conteudo'],
    )

@task
def planejamento_conteudo(self) -> Task:
    return Task(
        config=self.tasks_config['planejamento_conteudo'],
    )

@task
def escrita_blog_post(self) -> Task:
    return Task(
        config=self.tasks_config['escrita_blog_post'],
    )

@task
def revisao_conteudo(self) -> Task:
    return Task(
        config=self.tasks_config['revisao_conteudo'],
    )

@task
def salva_conteudo(self) -> Task:
    return Task(
        config=self.tasks_config['salva_conteudo'],
        tools=[file_write]
    )
```

```
@crew
def crew(self) -> Crew:
    return Crew(
        agents=self.agents,
        tasks=self.tasks,
        verbose=True,
    )
```

Podemos ver que estamos utilizando 3 ferramentas: - **SerperDevTool**: Faz uma busca no Google para um determinado assunto, encontrando assim páginas na web que possam ser relevantes. - **ScrapeWebsiteTool**: É responsável por acessar essas páginas web e fazer o download do conteúdo delas, permitindo que os modelos leiam seus conteúdos. - **FileWriterTool**: Pode escrever um arquivo no computador onde a aplicação está rodando.

Já demos passos importantes para transformar o gerador de blog posts em uma ferramenta realmente útil!

## 13. Criando tools personalizadas para CrewAI

Fala, pessoal! Estamos chegando ao final do nosso curso e considero que já aprendemos todas as bases do CrewAI. Agora quero passar um conceito um pouco mais avançado, que exige um pouco mais de conhecimento de programação: a criação de Tools próprias.

Como vocês viram, já existem diversas opções de ferramentas disponíveis na biblioteca para utilizarmos. No entanto, sabemos que a realidade é sempre mais complexa do que planejamos. Como programadores, precisamos usar a capacidade que a programação Python nos dá para nos adaptarmos às situações.

Saber criar uma Tool personalizada nos dá autonomia. Assim, deixamos de depender dos desenvolvedores da biblioteca e de restringir nossas aplicações ao que já foi criado, passando a nos tornarmos, de fato, criadores!

### Criando e utilizando Tools em CrewAI

Vou passar um passo a passo completo sobre como criar Tools. Não é nada tão complexo, mas vai exigir de vocês um pouco de atenção.

Para que o modelo consiga entender as nossas ferramentas, é importante descrever bem elas, tanto adicionando docstrings (aquelas explicações que vêm logo após a definição de uma função) quanto nos parâmetros de descrição.

Outro ponto importante é que precisamos cuidar bem dos nomes que damos às nossas ferramentas e aos argumentos delas, pois o modelo precisa ler aquela função, aquele argumento e suas definições para entender o que ela faz. Se o modelo não entender, ele não utilizará a ferramenta!

### Função que será usada como exemplo

Como exemplo, vamos criar uma ferramenta para buscar a cotação atual de uma ação utilizando a biblioteca Yahoo Finance.

Antes de criarmos a tool propriamente, vamos criar uma função em Python que faz uma chamada a api do Yahoo Finance e retorna um dicionário com a informação solicitada.

```
import yfinance as yf

def obter_cotacao_atual(acao: str):
    """
    Obtém a cotação mais recente de uma ação.

    :param acao: Código da ação (ex: "AAPL" para Apple, "PETR4.SA" para Petrobras)
```

```
:return: Dicionário com o preço atual e o horário da última atualização.
"""
try:
    ticker = yf.Ticker(acao)
    dados = ticker.history(period="1d") # Obtém os dados do último dia

    if not dados.empty:
        ultima_linha = dados.iloc[-1]
        return {
            "acao": acao,
            "preco_atual": ultima_linha["Close"],
            "ultima_atualizacao": ultima_linha.name.strftime("%Y-%m-%d %H:%M:%S"),
        }
    else:
        return {"erro": "Nenhum dado encontrado para a ação fornecida."}
except Exception as e:
    return {"erro": str(e)}

# Exemplo de uso:
cotacao = obter_cotacao_atual("AAPL")
print(cotacao)
```

Atenção! Isto ainda não é uma tool. Precisamos convertê-la a um formato que seja compreensível por um modelo dentro do framework CrewAI, e é o que veremos na sequência.

### Criando Tools com BaseTool

Vamos utilizar a classe `BaseTool` para criar nossa ferramenta. Ela deve seguir o seguinte formato, sendo que a execução da ferramenta se dará dentro do método `_run`.

```
from crewai.tools import BaseTool

class MinhaFerramentaPersonalizada(BaseTool):
    name: str = "Nome da minha ferramenta"
    description: str = "O que esta ferramenta faz. Isso é vital para uma utilização eficaz."

    def _run(self, argumento: str) -> str:
        # A lógica da sua ferramenta aqui
        return "Resultado da ferramenta"
```

É importante que o nome da ferramenta já indique seu propósito, além de ser relevante escrever boas descrições. O retorno do método `_run` deve obrigatoriamente ser uma string!

Aplicando para o nosso exemplo de função utilizando o Yahoo Finance, temos o seguinte:

```
from crewai.tools import BaseTool

import json
import yfinance as yf
```

```
class RetornaCotacaoAtual(BaseTool):
    name: str = "Retorna cotação atual"
    description: str = "Obtém a cotação mais recente de uma ação através da API do Yahoo
↳ Finance."

    def _run(self, acao: str) -> str:
        try:
            ticker = yf.Ticker(acao)
            dados = ticker.history(period="1d") # Obtém os dados do último dia

            if not dados.empty:
                ultima_linha = dados.iloc[-1]
                return json.dumps({
                    "acao": acao,
                    "preco_atual": ultima_linha["Close"],
                    "ultima_atualizacao": ultima_linha.name.strftime("%Y-%m-%d %H:%M:%S"),
                })
            else:
                return json.dumps({"erro": "Nenhum dado encontrado para a ação fornecida."})
        except Exception as e:
            return json.dumps({"erro": str(e)})
```

Você pode perceber que modificamos a saída da função ao utilizar o método `json.dumps`, que transforma um dicionário (tipo de saída original) em uma string!

### Descrevendo os argumentos

Para garantir que os argumentos da Tool sejam bem compreendidos pelo modelo, podemos utilizar o parâmetro `args_schema` para declará-los explicitamente, da seguinte forma:

```
from pydantic import BaseModel, Field
```

```
class RetornaCotacaoAtualInput(BaseModel):
    """Inputs da Tool RetornaCotacaoAtual."""
    acao: str = Field(..., description='Código da ação (ex: "AAPL" para Apple, "PETR4.SA" para
↳ Petrobras)')
```

E podemos fornecer este esquema à Tool utilizando o parâmetro `args_schema`, da seguinte forma:

```
from typing import Type
```

```
class RetornaCotacaoAtual(BaseTool):
    name: str = "Retorna cotação atual"
    description: str = "Obtém a cotação mais recente de uma ação através da API do Yahoo
↳ Finance."
    args_schema: Type[BaseModel] = RetornaCotacaoAtualInput

    def _run(self, acao: str) -> str:
        # Continuação da função ....
```

### Juntando tudo

Ao final, temos o seguinte código:

```
from typing import Type
from crewai.tools import BaseTool
from pydantic import BaseModel, Field

import json
import yfinance as yf

class RetornaCotacaoAtualInput(BaseModel):
    """Inputs da Tool RetornaCotacaoAtual."""
    acao: str = Field(..., description='Código da ação (ex: "AAPL" para Apple, "PETR4.SA" para Petrobras)')

class RetornaCotacaoAtual(BaseTool):
    name: str = "Retorna cotação atual"
    description: str = "Obtém a cotação mais recente de uma ação através da API do Yahoo Finance."
    args_schema: Type[BaseModel] = RetornaCotacaoAtualInput

    def _run(self, acao: str) -> str:
        try:
            ticker = yf.Ticker(acao)
            dados = ticker.history(period="1d") # Obtém os dados do último dia

            if not dados.empty:
                ultima_linha = dados.iloc[-1]
                return json.dumps({
                    "acao": acao,
                    "preco_atual": ultima_linha["Close"],
                    "ultima_atualizacao": ultima_linha.name.strftime("%Y-%m-%d %H:%M:%S"),
                })
            else:
                return json.dumps({"erro": "Nenhum dado encontrado para a ação fornecida."})
        except Exception as e:
            return json.dumps({"erro": str(e)})
```

E pronto, temos um exemplo completo e funcional de como criar uma Tool!

### Utilizando Tools do LangChain

Saber utilizar Tools nos dá flexibilidade, inclusive para integrar ferramentas de outros frameworks com o CrewAI, e é o caso do LangChain.

Podemos aproveitar as diversas ferramentas já desenvolvidas pela comunidade do LangChain e integrá-las aos nossos projetos. Para verificar as ferramentas disponíveis, você pode acessar esta página: [LangChain Tools](#).

### Integrando Ferramentas do LangChain ao CrewAI

Mas vamos ver, na prática, como podemos fazer isso. A estrutura será similar à criação de uma ferramenta personalizada, com a diferença de que a classe da ferramenta do LangChain será importada e o método `_run` apenas executará a Tool da seguinte forma:

```
import os
from dotenv import load_dotenv
from crewai import Agent, Task, Crew
from crewai.tools import BaseTool
from pydantic import Field
from langchain_community.utilities import GoogleSerperAPIWrapper

# Configure sua chave SERPER_API_KEY em um arquivo .env, por exemplo:
# SERPER_API_KEY=<sua chave da API>
load_dotenv()

class SearchTool(BaseTool):
    name: str = "Search"
    description: str = "Útil para consultas baseadas em busca. Use isso para encontrar
    ↪ informações atuais sobre mercados, empresas e tendências."
    search: GoogleSerperAPIWrapper = Field(default_factory=GoogleSerperAPIWrapper)

    def _run(self, query: str) -> str:
        """Executa a consulta de busca e retorna os resultados"""
        try:
            return self.search.run(query)
        except Exception as e:
            return f"Erro ao realizar a busca: {str(e)}"
```

No exemplo acima, estamos criando uma ferramenta de busca chamada `SearchTool`, que utiliza a classe `GoogleSerperAPIWrapper` do LangChain para buscar resultados no Google.

Notem que definimos a ferramenta `GoogleSerperAPIWrapper` do LangChain como um atributo da classe que estamos criando:

```
class SearchTool(BaseTool):
    #...
    search: GoogleSerperAPIWrapper = Field(default_factory=GoogleSerperAPIWrapper)
```

E depois podemos utilizar essa ferramenta dentro do nosso método `_run`:

```
def _run(self, query: str) -> str:
    #...
    return self.search.run(query)
    #...
```

Com essa simplicidade, temos acesso a todas as ferramentas do framework LangChain!

Aproveitem para dar uma olhada nas ferramentas existentes e comentem aqui conosco quais delas fazem sentido para vocês!



## 14. Finalizando o curso

Parabéns, meu amigo! Você chegou ao final do nosso curso “Criando Multiagent Systems com CrewAI”! Para encerrar, vamos fazer uma breve recapitulação para fixar as ideias que exploramos, além de te dar algumas dicas para o futuro.

### Introdução – Recapitulando o curso

Desde o início, nosso objetivo foi mostrar como quebrar problemas complexos em partes menores e mais fáceis de lidar. A ideia é simples: assim como em uma equipe, cada membro tem uma função específica e contribui para o sucesso do projeto.

Vimos como dividir as tarefas e organizar os **agents**, **tasks**, **crews**, **processes** e **tools** é fundamental para resolver desafios complicados com mais clareza e eficiência. **Esse é o propósito do framework CrewAI!**

---

### Recapitulação dos conceitos fundamentais

- **Agents** – Aprendemos que cada agente deve ter uma função bem definida, com um **role**, **goal** e **backstory** claros. Assim, cada “pessoa” da nossa crew sabe exatamente no que focar.
  - **Tasks** – Discutimos a importância de planejar direitinho as atividades: tasks de reflexão (como planejamento e revisão) mostram que dar um tempo para pensar pode render resultados bem melhores do que tentar fazer tudo de uma vez, sem pausas, no famigerado “zero-shot”.
  - **Crews** – Montar a crew é como montar um time competente: quando as interações e o fluxo de trabalho estão organizados, tudo funciona de forma harmoniosa e os resultados ficam muito mais incríveis.
  - **Tools** – As ferramentas expandem as capacidades dos nossos modelos. Seja para buscar dados, interagir com APIs ou até executar ações específicas, as tools – incluindo as que criamos personalizadas – são essenciais para transformar ideias em soluções reais.
-

## Aplicações práticas e exemplos

Ao longo do curso, mostramos exemplos práticos como a criação de um post de blog, a roteirização de vídeos para o YouTube e até como estruturar um projeto com arquivos YAML para produção. Esses exemplos foram pensados para evidenciar na prática como os conceitos teóricos se transformam em soluções funcionais. Se você aplicou esses exemplos ou se inspirou neles, já deu um grande passo para desenvolver suas próprias soluções!

---

## Dicas e direcionamentos para o futuro

Agora, quero deixar um conselho fundamental: **continue experimentando!**

Não existe um único jeito “certo” de montar uma solução. O que realmente conta é praticar, testar novas ideias e nunca ter medo de errar.

- Experimente criar novas crews e integrar mais fontes de dados com tools externas, como as do **LangChain**.
  - Aproveite a **AgentOps** para acompanhar a performance dos seus agentes em cenários reais; essa ferramenta é poderosa para identificar onde podemos melhorar.
  - Sempre revise e refine seus processos – o segredo está em modularizar e manter tudo bem organizado para facilitar a evolução dos seus projetos.
- 

## E pra finalizar

Lembre-se: dividir um problema complicado em etapas claras é uma habilidade que vai muito além do CrewAI. Essa técnica pode ser aplicada em várias áreas da programação e do desenvolvimento de software. Se destacar no mundo dos sistemas multiagentes depende de uma definição clara das responsabilidades e de uma integração eficiente entre os diversos componentes do sistema.

Parabéns por ter chegado até aqui! Espero que você se sinta motivado a continuar testando novas abordagens e, principalmente, a nunca parar de aprender e aprimorar suas ideias. Você está preparado para criar soluções cada vez mais inteligentes – siga em frente e use essa base que construímos juntos para alcançar voos ainda maiores!

Um grande abraço e até a próxima!